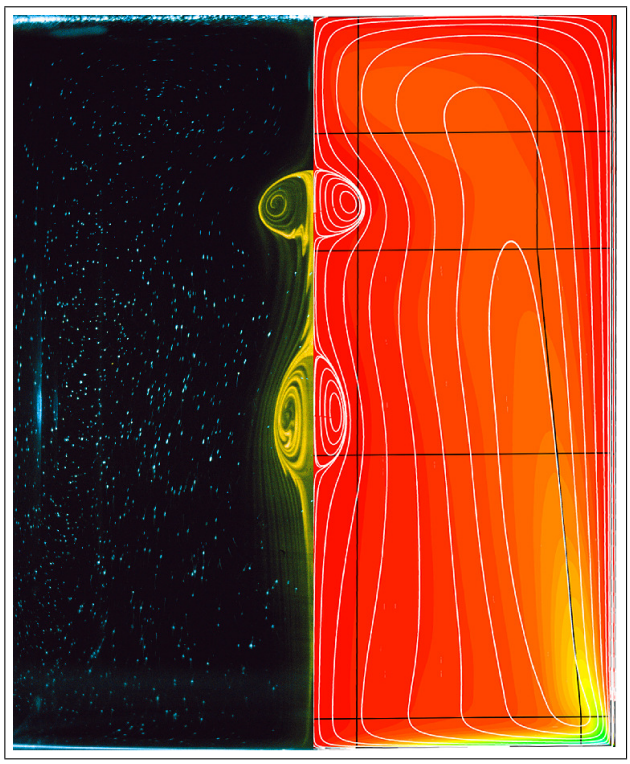
Using *Semtex*



H. M. Blackburn
Monash University

March 19, 2025
Semtex version 10

Contents

Chapter 1

Introduction

Semtex is a family of spectral element simulation codes, most prominently a code for direct numerical simulation of incompressible flow. The spectral element method is a high-order finite element technique that combines the geometric flexibility of finite elements with the high accuracy of spectral methods. The method was pioneered in the mid 1980's by Anthony Patera at MIT (Patera; 1984; Korczak and Patera; 1986). *Semtex* uses isoparametrically mapped two-dimensional quadrilateral elements, the classic Gauss–Lobatto–Legendre 'nodal' shape function basis, and continuous Galerkin projection. Extension to three-dimensional capability is achieved using Fourier expansions in an orthogonal direction. Algorithmically the code is similar to Ron Henderson's *Prism* (Henderson and Karniadakis; 1995; Karniadakis and Henderson; 1998; Henderson; 1999), but with some differences in design, and lacks mortar element capability. A notable extension is that *Semtex* can solve problems in cylindrical as well as Cartesian coordinate systems (Blackburn and Sherwin; 2004; Blackburn et al.; 2019).

1.1 Numerical method

Some central features of the spectral element method are

Orthogonal polynomial-based shape functions Spectral accuracy is achieved by using tensor-product Lagrange interpolants within each element, where the nodes of these shape functions are placed at the zeros of Legendre polynomials mapped from the canonical domain $[-1,+1]\times[-1,+1]$ to each element. In one spatial dimension, the resulting Gauss–Lobatto–Legendre interpolant which is unity at one of the $N+1$ Gauss–Lobatto points x_j in $[-1,+1]$ and zero at the others is

$$\psi_j(x) = \frac{1}{N(N+1)L_N(x_j)}\frac{(1-x^2)L'_N(x)}{x_j - x}. \tag{1.1}$$

For example, the family of sixth-order GLL Lagrange interpolants is shown in figure 1.1. In smooth function spaces it can be shown that the resulting interpolants converge exponentially fast (faster than any negative integer power of N) as the order of the interpolant is increased. See Canuto et al. (1988), §§ 2.3.2 and 9.4.3 or Canuto et al. (2006) § 5.4.

Standard finite element isoparametric mapping Two-dimensional element shape functions are constructed as tensor products of one-dimensional shape functions. Non-rectangular element shapes, if required, are developed using isoparametric mappings between physical (x,y) space and master element (r,s) space on the domain $[-1,+1]\times[-1,+1]$, as illustrated in figure 1.2.

Gauss–Lobatto quadrature Gauss–Lobatto quadrature is used for approximating elemental integrals: the quadrature points reside at the nodal points, which enables fast tensor-product

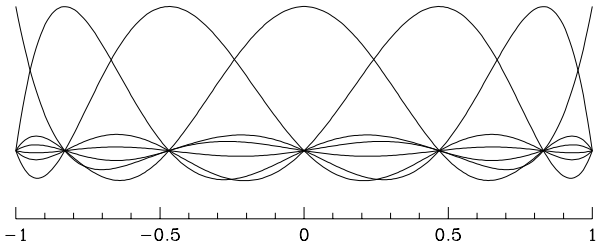


Figure 1.1: The family of sixth-order one-dimensional GLL Lagrange shape functions on the master domain $[-1, +1]$.

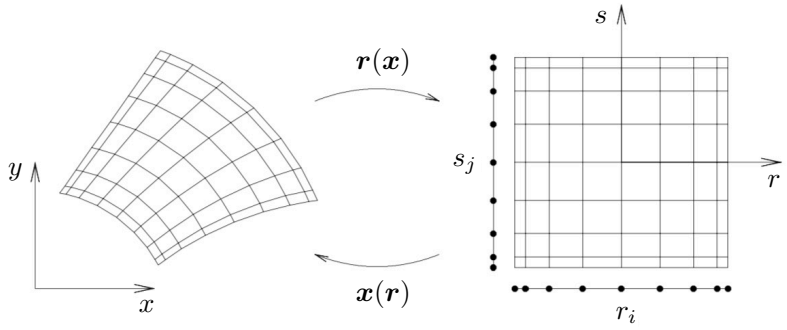


Figure 1.2: Shape functions on two-dimensional elements are constructed using tensor products of one-dimensional shape functions, incorporating an isoparametric mapping from (x, y) to master element (r, s) space.

techniques to be used for iterative matrix solution methods. Gauss–Lobatto quadrature on the nodal points conveniently produces diagonal mass matrices when Lagrange interpolants are used as basis functions.

Static condensation Direct matrix solutions are sped up by using static condensation coupled with bandwidth reduction algorithms to reduce storage requirements for assembled system matrices.

While the numerical method is very accurate and efficient, it also has the advantage that complex geometries can be accommodated by employing unstructured conforming meshes. The vertices of spectral elements meshes can be produced using finite-element mesh generation procedures, or any other method (for *Semtex*, only meshes with quadrilateral elements are accepted).

Time integration employs a backwards-time differencing scheme described by Karniadakis et al. (1991), more recently classified as a velocity-correction method by Guermond and Shen (2003). One can select first, second, or third-order time integration, but second order is usually a reasonable compromise, and is the default scheme. Equal-order interpolation is used for velocity and pressure (see Guermond et al.; 2006).

As of *Semtex* V8, the 'alternating skew symmetric' form (Zang; 1991) is the default for construction of nonlinear terms in the Navier–Stokes equations (faster and just as robust as full skew symmetric, which is still an option), and no dealiasing of product terms is carried out for either serial or parallel operations. As an aid to robust operation at high Reynolds numbers, 'spectral vanishing viscosity' (Xu and Pasquetti; 2004) can easily be enabled by setting appropriate control tokens. A significant additional novelty of *Semtex* V9.3 is the option of robust energy-stable open boundary conditions (Dong; 2015), which alleviate much of the numerical stability problem associated with inflows that occur at the outflow boundary. These also allow ingestion of flow without causing blow-ups. As of *Semtex* V9, DNS variables may optionally contain a scalar variable in addition to velocity

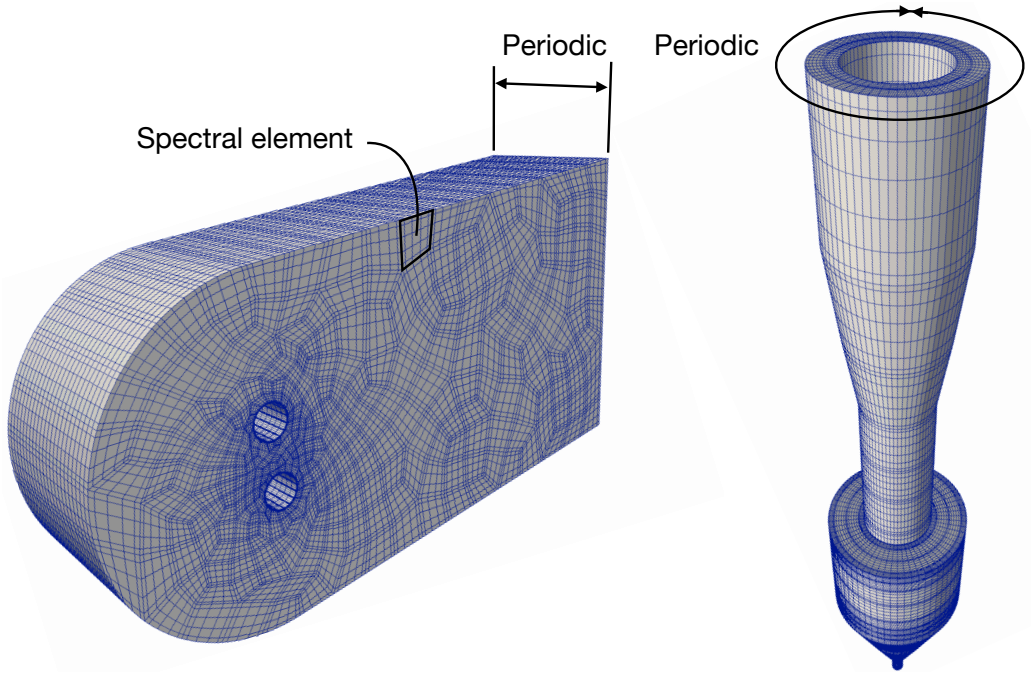


Figure 1.3: *Semtex* can solve either elliptic or incompressible Navier–Stokes problems in domains which are either two-dimensional or made three-dimensional by extrusion of a two-dimensional domain in a periodic direction. Either Cartesian (left) or cylindrical (right) coordinate systems can be used.

components and pressure. If requested (by running `dns -f`), evolution of the scalar can be obtained in a frozen, pre-supplied velocity field, i.e. as solution of an advection–diffusion problem.

As suggested in figure 1.3, *Semtex* can solve problems in two-dimensional domains or in three-dimensional domains that can be obtained from arbitrary two-dimensional domains by extrusion in an orthogonal direction in which the solution fields are periodic; often such problems are referred to as $2\,{}^1\!/_2$-dimensional. Quite a large number of fundamental problems in mechanics can be tackled using this level of geometric complexity, but if you need genuinely three-dimensional geometries then look elsewhere. While parallel execution is supported, the method is only parallel across the homogeneous/extrustion/Fourier direction, so each process has to be able to accommodate at least two two-dimensional data planes and associated overheads. Sometimes this design restriction is significant. Code performance is typically quite efficient (and fast) up to some hundreds or perhaps thousands of processors, but this is problem-dependent.

1.2 Implementation

The top level of the code is written in C++, with calls to C and Fortran library routines, e.g. BLAS and LAPACK. The original implementation for two-dimensional Cartesian geometries was extended to three dimensions using Fourier expansion functions for spatially-periodic directions in Cartesian and cylindrical spaces. Concurrent execution is supported, using MPI as the basis for interprocess communications, and the code has been run on a wide variety of conventional multiprocessor machines. Basically it ought to work with little trouble on any contemporary Unix system. GPUs are not supported, nor is OpenMP. The code is unlikely to benefit much, if at all, from multi-threaded execution, and we generally suggest that multi-threaded execution be disabled.

There have been various code extensions that are not part of the base distribution. These include dynamic and non-dynamic LES (Blackburn and Schmidt; 2003), simple power-law type non-Newtonian rheologies (Rudman and Blackburn; 2006), accelerating frame of reference coupling for

aeroelasticity (Blackburn and Henderson; 1996, 1999; Blackburn et al.; 2000; Blackburn; 2003), solution of steady-state flows via Newton–Raphson iteration (Blackburn; 2002)

However, linear stability analysis (Blackburn; 2002; Blackburn and Lopez; 2003a,b; Blackburn et al.; 2005; Sherwin and Blackburn; 2005; Elston et al.; 2006; Blackburn and Sherwin; 2007) and optimal transient growth analysis (Blackburn et al.; 2008) are released as an additional open-source code base (called *Dog*), with a separate user guide called 'Working *Dog*'.

1.3 Further reading

The numerical techniques used by *Semtex* are summarised in Blackburn et al. (2019). A good introduction to (low-order) finite element methods is provided by Hughes (1987). The most comprehensive references on spectral methods in general are Gottlieb and Orszag (1977), Canuto et al. (1988, 2006). The first papers by Patera (1984) and Korczak and Patera (1986) provide a good introduction to spectral elements, although some implementation details changed with time and Maday and Patera (1989) is more reflective of the methods used in *Semtex*. The adoption of Fourier expansions to extend the method to three spatial dimensions is discussed by Amon and Patera (1989), Karniadakis (1989) and Karniadakis (1990). The use of spectral element techniques in cylindrical coordinates is dealt with in Blackburn and Sherwin (2004). The book by Funaro (1997) provides useful information and further references. Overviews and some applications appear in Karniadakis and Henderson (1998); Henderson (1999). The definitive reference is now the book by Karniadakis and Sherwin (2005), but you will also find the text by Deville, Fischer and Mund (2002) useful for alternative explanations and views. More recently, the book by Canuto et al. (2007) provides both theory and applications of spectral as well as spectral element methods in fluid dynamics.

Chapter 2

Starting out

2.1 Computational environment

It is assumed you are using some version of Unix (which includes Mac OS X), with a development system that includes C, C++ and Fortran compilers, bison (or yacc), make, and optionally cmake. For post-processing, *SuperMongo* and *Tecplot* would be nice to have but are not essential to get up and running, and VTK-based visualisation tools such as *VisIt* or *ParaView* can alternatively be used in place of *Tecplot*.

Instructions for building and testing the codes are given in the accompanying `StartHere.txt` and `README.md` files in the top-level directory. The rest of the present document assumes that you are able to build a working set of executables and place them in your `PATH`.

2.2 Equations to be solved

The central solvers provided by the *Semtex* package are

`elliptic`	for elliptic (Laplace, Poisson, Helmholtz) problems,
`dns`	for time-varying incompressible Navier–Stokes problems, with an optional scalar,

and (if MPI is present) their equivalent multi-process versions `elliptic_mp` and `dns_mp`, which can speed up solution of three-dimensional problems (see § 3.9.2). There are also a number of associated pre- and post-processing utilities.

Elliptic equations dealt with by *Semtex* are in general of Helmholtz type;

$$\nabla^2 c - \lambda^2 c = f, \tag{2.1}$$

where λ is a real constant and f is in general a function of space; if $\lambda = 0$ we have Poisson's equation while if also $f = 0$ we have Laplace's equation. These equations are solved via a Petrov–Galerkin method using standard finite-element techniques. While elliptic equations may be of less interest to many readers than the Navier–Stokes equations, it is simple to provide an elliptic solver since it underlies the time-splitting approach adopted for tackling the Navier–Stokes equations, i.e. basically as a sequence of solutions to elliptic scalar equations in each timestep.

The incompressible Navier–Stokes equations are

$$\partial_t \boldsymbol{u} + \boldsymbol{N}(\boldsymbol{u}) = -\nabla P + \nu \nabla^2 \boldsymbol{u} + \boldsymbol{f} \quad \text{with} \quad \boldsymbol{\nabla \cdot u} = 0; \tag{2.2}$$

$P = p/\rho$ is sometimes called the modified pressure and $\nu = \mu/\rho$ is the kinematic viscosity, while $\boldsymbol{f}$ represents body force per unit mass; see § 4.13 for a description of the forms of $\boldsymbol{f}$ implemented in the code. The nonlinear terms $\boldsymbol{N}(\boldsymbol{u})$ can be represented in two (or more) ways which are equivalent in the continuous setting but have somewhat different behaviour in the discrete setting:

either in the 'non-conservative' form $\boldsymbol{u}\cdot\boldsymbol{\nabla u}$ or the 'skew-symmetric' form $[\boldsymbol{u}\cdot\boldsymbol{\nabla u}+\boldsymbol{\nabla}\cdot\boldsymbol{uu}]/2$. While both forms are provided, generally we use the skew-symmetric form since compared to the non-conservative form it tends to be more robust (has better energy conservation properties), or a simplified/cheaper form called the 'alternating skew-symmetric' which alternates between using $\boldsymbol{u}\cdot\boldsymbol{\nabla u}$ and $\boldsymbol{\nabla}\cdot\boldsymbol{uu}$ on successive timesteps; this proves to be almost as robust as full skew-symmetric but has a computational cost equivalent to the non-conservative form. We can optionally demand that $\boldsymbol{N}(\boldsymbol{u})=0$ in which case we have the (unsteady) Stokes equations. If an advected scalar c is present, the Navier–Stokes equations are augmented by the advection–diffusion equation

$$\partial_t c+\boldsymbol{C}(c)=\alpha\nabla^2 c, \tag{2.3}$$

where $\alpha=\nu/Pr$ (Pr being the Prandtl number). In this case again, $\boldsymbol{C}(c)$ can take non-conservative form $\boldsymbol{u}\cdot\boldsymbol{\nabla}c$, skew-symmetric form $[\boldsymbol{u}\cdot\boldsymbol{\nabla}c+\boldsymbol{\nabla}\cdot\boldsymbol{u}c]/2$, the alternating equivalent or indeed $\boldsymbol{C}(c)=0$ according to what is requested for the momentum equations. It is possible to run `dns` in a mode which uses a 'frozen' velocity field $\boldsymbol{u}$, in which case just the advection–diffusion equation for c is integrated forward in time.

Numbers of velocity components and spatial dimensions: for Navier–Stokes type problems, *Semtex* can solve problems which are (a) two-dimensional and two-component, (b) two-dimensional and three-component, or three-dimensional and three-component (a.k.a. 2D2C, 2D3C, 3D3C).

As with many codes used to solve the unsteady Navier–Stokes equations, diffusion-type terms (those involving ∇^2) are dealt with implicitly in time, while the nonlinear-type terms $\boldsymbol{N}$ and $\boldsymbol{C}$ are dealt with explicitly, so that stable time integration generally requires a time-step restriction of CFL type.

2.3 Mesh resolution — and design

This is such a large area of discourse that we can't hope to adequately cover it here; the following brief remarks are intended as an introduction only.

Since it employs high-order finite element methods in the (x,y) plane, *Semtex* offers the choice of element-size-based refinement (so-called h-refinement) or polynomial-order-based refinement (so-called p-refinement) in attempting to converge a solution (it is an hp-type method). The basis polynomials share convergence properties with (orthogonal) Legendre polynomials and the underlying goal of mesh refinement is to achieve exponential ('spectral') convergence of solutions with respect to polynomial order; this typically commences when there are of order π polynomials per 'wavelength' of solution variation (Gottlieb and Orszag; 1977).

We should point out that p-refinement is very straightforward in *Semtex*; different polynomial orders can be selected simply by varying the token `N_P`, the number of mesh points along the edge of every element in the problem session file (see e.g. §2.5.5): the one-dimensional polynomial order $p=$ `N_P` -1. On the other hand, carrying out h-refinement will require a completely new session file to be produced, which may imply quite a bit more work. We also note that computational work per timestep, typically dominated by forming the nonlinear terms of the Navier–Stokes equations, tends to scale like `N_P`^{2} (and, of course, the number of elements). Finally, *Semtex* tends to be most efficient at moderate polynomial orders (e.g. 4 to 13); it is generally not a good idea to choose either very low, or very high, values of p.

A rule-of-thumb for mesh design based on the remarks above is to aim to have any 'significant' rapid variation in solution behaviour covered by one or two elements (the goal of h-refinement), and then carry out whatever p-refinement is desired to get well-converged results. The path of least resistance for two-dimensional mesh design is generally to produce a session file which one guesses is quite well resolved as far as element sizes go, then rely on converging the solution by starting on the low end of the p range and then increasing p to higher, yet moderate, values in order to commence exponential convergence. A few iterations of mesh design may be required when tackling a new problem.

In the z-direction, *Semtex* assumes the solution is periodic and uses Fourier expansions (with a 2–3–5-prime-factor FFT from Temperton; 1992). Choice of spanwise length scale L_z is controlled by the token `BETA`, where $\beta = 2\pi/L_z$, while resolution is controlled by token `N_Z`, the number of planes of real data in the z direction (the number of complex Fourier modes is then `N_Z`$/2$). Convergence follows standard rules for Fourier-spectral methods. See § 3.3.3 for discussion regarding possible values for `N_Z`.

2.4 Files

Semtex uses a text input file which describes the mesh, boundary conditions and other problem parameters. We call this a session file and typically it has no root extension. It is written in a format loosely patterned on HTML, which we have called FEML (for Finite Element Markup Language) — alternatively you could consider FEML as a cut-down version of XML. FEML is 'just enough' to compactly describe spectral element problems to the solvers. There are a number of example session files provided in the mesh subdirectory; some of these are used in regression testing (and/or described in the following document). Files other than session file have standard extensions:

`session.fld`	Solution/field file. Binary format by default, but with a 10-line ASCII header.
`session.rst`	Restart file, same format as field file. Read in to initialise solution if present.
`session.chk`	Intermediate solution checkpoint/restart files.
`session.avg`	Averaged results. Read back in for continuation (over-written).
`session.his`	History point data.
`session.flx`	Time series of pressure and viscous forces integrated over the `wall` boundary group.
`session.mdl`	Time series of kinetic energies in solution Fourier modes.
`session.par`	Used to define initial particle locations.
`session.trk`	Integrated particle locations.

When writing a new session file it is prudent to run `meshpr` (and/or `meshpr -c`) on it before trying to use it for simulations, since `meshpr` will catch most of the easier-to-make geometric anomalies. You can also plot up the results using *SuperMongo* or other utility (such as `meshplot`) as a visual check.

NB: the 10-line ASCII header of any field-type file can quickly be viewed using the Unix `head` utility, regardless of the format of following field data.

2.5 Structure of a session file

More details and examples for such files are provided below and in the next chapter, but a brief outline is that a session file is an ASCII text file with a number of sections that should each appear once, but which can be supplied in arbitrary order. Any line whose first character is # is taken as a comment line; these can appear in arbitrary locations within a session file. It is generally safe to use spaces, tabs, or new lines as white space in a session file, where such space is permitted; within function strings to be interpreted by the internal function parser, it is not. In a session file, integer indices that number specific entities such as `NODES` and `ELEMENTS` are indexed starting from 1: this is also true of related warning or error messages that *Semtex* codes may issue, though within the code itself, such indices are if necessary adjusted to follow the standard C convention of being indexed starting from 0. Much of the structure of a session file is actually free-format, but it is generally safest to assume that collections of data which are usually shown on a single line in the examples should not extend over multiple lines.

As in HTML, sections are demarcated using paired opening (<name>) and closing (</name>) tags. The opening tag will sometimes require a numeric attribute, e.g. `<NODES NUMBER=10>`. The

minimal set of sections required for a session file to validly describe a domain in which a solution can be obtained is: NODES, ELEMENTS and SURFACES: NODES describe the corner vertices of ELEMENTS, while SURFACES describe what happens on the outer boundaries of the domain. Sides of all ELEMENTS — which run between NODES — must either mate with the side of another ELEMENT or be dealt with in the SURFACES section.

2.5.1 NODES

This section describes the location of the NODES which position ELEMENT corner vertices in (x, y) space. The attribute NUMBER is required in the opening tag; this should match the number of NODES which follow. This can exceed the number of unique NODES which are actually required for all the element vertices, i.e. spare/unused NODES are allowed, but they have to be included in the NUMBER count. The minimum required NUMBER of NODES is four, i.e. for a single ELEMENT. Each line of the NODES section has four entries: an integer numeric index — mostly used by human readers of the file, since it is ignored by the code — followed by three numbers which give the location of each NODE in (x, y, z) space (despite the fact that only the (x, y) coordinates are currently used by *Semtex*). Different NODES can take the same position in (x, y) space, and in fact, if you want to make a zero-width slit boundary within a mesh, differently-indexed NODES at identical locations will be required. Example, with just enough NODES for a single ELEMENT:

```
<NODES NUMBER=4>
  1       0.0       0.0       0.0
  2       1.0       0.0       0.0
  3       0.0       1.0       0.0
  4       1.0       1.0       0.0
</NODES>
```

It is often convenient to move and rescale the nodal locations declared in this section. This may be achieved simply by setting TOKENS called X_SCALE, X_SHIFT, Y_SCALE and Y_SHIFT. The convention is that the locations are first shifted and then scaled.

2.5.2 ELEMENTS

The ELEMENTS section also requires a NUMBER attribute, which gives the number of ELEMENTS that follow. The minimum required NUMBER of ELEMENTS is one. *Semtex* only presently allows quadrilateral elements, though these may have curved edges; despite this restriction, each element is listed with an opening <Q> tag and a closing </Q> tag. The data that describe and element are: a unique integer identifier, followed by the opening tag <Q>, four integer NODE identifiers, terminated by the closing tag </Q>. The ordering of these NODES must be such that their locations traverse a patch of (x, y) space in a counter-clockwise sense, but is otherwise arbitrary.

The sides of an ELEMENT are taken as running between the vertex nodes with a 1-based indexing: the side between the first and second NODES is number 1, the side between the second and third NODES is number 2, and so on. This information is relevant in the following SURFACES section. Sides of distinct elements should of course not cross one another, though this is not actually enforced in the code. What *is* checked is that for all elements, each side either matches the side of another element, as determined using the NODE indices of all ELEMENTS, or has valid SURFACE information supplied. This ensures that the (x, y) region of a two-dimensional solution domain, though possibly multiply-connected, is completely covered by the spectral element mesh and has appropriate boundary condition information. A one-ELEMENT example based on the previous set of NODES follows:

```
<ELEMENTS NUMBER=1>
  1  <Q>  4  3  1  2  </Q>
</ELEMENTS>
```

2.5.3 SURFACES

Any ELEMENT side that does not mate to the side of another ELEMENT based on its NODES requires a link to boundary condition information elsewhere in the session file or be paired up with another element side (to provide a periodic connection). The SURFACES section deals with these requirements. Again the opening tag must include a numeric attribute that matches the following number of surface specifications.

Each surface is described by: a single integer index (again, mostly for convenience of human readers), followed by a pair of integers that give element and side numbers for this SURFACE (these relate back to information given in the ELEMENTS section), and then a tag-delimited list which provides either a single-character boundary condition GROUP (delimited by <B> and </B>) or a pair of integers that are the element and side of a periodic connection (delimited by <P> and </P>. Note that a periodic connection effectively deals with a pair of sides, and the reverse connection should not also appear. Here is an example for a single ELEMENT with two boundary conditions and a periodic self-connection:

```
<SURFACES NUMBER=3>
  1  1  1  <B>  w  </B>
  2  1  3  <B>  t  </B>
  3  1  2  <P>  1  4  </P>
</ELEMENTS>
```

2.5.4 FIELDS

This brief section nominates the list of solution variables (i.e. those which satisfy a partial differential equation and for which we must provide boundary conditions). The FIELDS are given standard single-character name-tags: u, v, w and p for (x, y, z) velocity components and pressure, and c for scalar. If we are solving an elliptic equation, only c is required; if solving the Navier–Stokes equations, at least u, v and p must be present (for a 2D2C solution), and c can also appear (as a transported scalar field). Generally, velocity components and (optional) scalar should appear in the list before the pressure. Example for a 2D2C flow with scalar:

```
<FIELDS>
  u  v  c  p
</FIELDS>
```

2.5.5 TOKENS

In this section, TOKENS that have significance either directly to the solver or for use elsewhere in the session file may be defined. This section *is not* free-form: each TOKEN must be defined on a separate line in the form

```
TOKEN = string_evaluating_to_value
```

The string following the equals sign should not contain white space and may be a maximum of 2048 characters in length. It can either directly supply a numeric value, or be a string that may be parsed to deliver a numeric value. The function parser is patterned on the hoc3 program in Kernighan and Pike (1984) which uses yacc, and has many standard math functions such as $\exp$, $\cos$ and $\tan^{-1}$ as well as standard arithmetic operations $+$, $-$, $*$ and $/$. TOKENS can be defined in terms of standard/inbuilt TOKENS or ones which have been defined earlier in the list. To see all the predefined TOKENS and functions, check the *Semtex* utility "calc -h" — calc uses the same parser to evaluate expressions given on the command line.

Below is a simple example in which (in order) cylindrical coordinates are selected, the time step, D_T, is set, as is the time-stepping order N_TIME, the number of points along the edge of every element, N_P, following which a value for a new TOKEN, Re, is obtained by parsing a string and used to set the kinematic viscosity KINVIS as used by the dns solver. (On its own, Re is not directly significant to the solver.) All such TOKENS may be used subsequently in the BCS or FORCE sections.

```
<TOKENS>
  CYLINDRICAL = 1
  D_T         = 0.001
  N_TIME      = 1
  N_P         = 7
  Re          = (30/exp(1.6))^3.
  KINVIS      = 1/Re
</TOKENS>
```

This would result in the token KINVIS taking the value 0.0450039 (to 6 sf.). Note that TOKENS is evaluated before any other section of a session file, so that its definitions have global scope.

2.5.6 GROUPS

This is another brief section but one that requires a numeric parameter and which is used in defining boundary conditions. A GROUP associates a single-character name and a string. The string can potentially be used by more than a single GROUP, thus linking them together. The name is also used both in the BCS section and the SURFACES section, so provides a short-hand way of linking BCS to SURFACES. At first this seems a redundant indirection; why not just have SURFACES and BCS?, but it is useful because of the ability to join up GROUPS of BCS via a common string, which may have significance to the solver: e.g. all BCS in the wall group will have normal and tangential tractions evaluated, integrated and printed to file session.flx, even though the BCS involved may differ.

```
<GROUPS NUMBER=4>
  1  h  wall
  2  c  wall
  3  a  axis
  4  v  speed
</GROUPS>
```

As will be elaborated in chapter 3, the string axis is also significant to the solver if cylindrical coordinates are chosen by setting CYLINDRICAL = 1 in TOKENS, while the string open is required for a GROUP associated with a set of boundary conditions of energy-stable open type (Dong; 2015).

2.5.7 BCS

Data in this section links a GROUP character with a brief description of the set of boundary conditions to be applied for solution of each FIELD. Again, a NUMBER attribute is required in the opening tag. For each set of BCS, we must supply: an integer index, a GROUP character, and another integer for the number of BC descriptors that follow (which should match the number of solution FIELDS). Subsequent to that are the actual descriptors of the BCS which are to be applied to each FIELD in the relevant GROUP. While at first it may seem that there are apparently a variety of BC types which may be selected, in practice the solver is only capable of applying either Dirichlet, Neumann or mixed (Robin) boundary conditions for any FIELD, but sometimes these are set automatically within the code. Here is an introductory example:

```
<BCS NUMBER=1>
  1  v  4
        <D> u = 1-exp(LAMBDA*x)*cos(2*PI*y)             </D>
        <D> v = LAMBDA/(2*PI)*exp(LAMBDA*x)*sin(2*PI*y) </D>
        <D> w = 0.0                                     </D>
        <H> p                                           </H>
</BCS>
```

We see that for each FIELD variable a boundary condition is set. For Dirichlet (value) boundary conditions the tag used is <D>; for Neumann (gradient) boundary conditions the tag is <N>; for mixed boundary conditions the tag is <M>. In the example, Dirichlet conditions are supplied for FIELDS u, v and w; these can either be set by parsing a string (which should contain no white space), or directly

with a numeric value. In the case of parsing a string, we can use TOKENS and/or space/time variables x, y, z and t; the string supplied is re-parsed every timestep as the code runs and t updates. We see that for the pressure, p, we have given the special tag <H> to denote a "high-order" boundary condition, which is actually an internally-computed Neumann type (see Karniadakis et al.; 1991).

For more details regarding boundary condition types, see § 3.7 below.

2.5.8 CURVES

Element edges do not have to be straight; the user can alternatively select circular arcs or a spline fitted through data supplied in external files. The information is supplied to the solver in the CURVES section.

```
<CURVES NUMBER=2>
  1  4  3  <ARC> 1.0 </ARC>
  2  1  1  <SPLINE> splinepoints.dat </SPLINE>
</CURVES>
```

The first number on each line is just a numeric label for convenience of human readers. For each numbered CURVE we also provide the element and side number to which it applies (in the first of the above, these numbers are 4 and 3 respectively) followed by a tagged section which nominates the kind of curve.

For a circular ARC, we must give the radius as a signed number: positive values denote convex arcs (relative to the element centroid) while negative values denote concave arcs. Such curves do not have to be on the external boundary of a mesh, but could be internal: in this case the user must also supply corresponding information for the mating element edge. The code does not check that such curves conform (e.g. with matching $\pm$ radii); that is up to the user.

For a SPLINE, the supplied parameter is the name of an ASCII text file that supplies (x, y) locations of points (one pair per line) through which a cubic spline with natural end conditions is fit. Spectral element knot points are interpolated along this splined curve. (A mesh NODE which might not lie exactly on the spline is moved slightly by projection along its other edge, until it lies on the spline to within a tolerance of 1E-6.) If no path is supplied as part of the name, the file is assumed to reside in the directory from which execution is initiated. The same data file can be named for a number of curved edges. See § 3.1.1 below for further information and discussion relating to SPLINE curves, and consult the *Dog* user guide for an example.

2.5.9 FORCE

This optional section declares various kinds of body-force information for Navier–Stokes momentum equations. Please consult § 4.13 for more detail.

2.5.10 USER

The principal purpose of the USER section is in specifying solution values which are parsed and written out as a field dump using the compare utility. These could be e.g. an analytical solution or an initial condition such as a solid-body rotation. If they are an analytical solution, another utility (rstress) could subsequently be used to subtract the computed solution — this technique is used in regression testing. Example:

```
<USER>
  u = -cos(PI*x)*sin(PI*y)*exp(-2.0*PI*PI*KINVIS*t)
  v =  sin(PI*x)*cos(PI*y)*exp(-2.0*PI*PI*KINVIS*t)
  p = -0.25*(cos(TWOPI*x)+cos(TWOPI*y))*exp(-4.0*PI*PI*KINVIS*t)
</USER>
```

2.5.11 HISTORY

This section controls the output of history point data at specific points in the domain to an ASCII file called session.his. Output is made every IO_HIS integration steps. Solution data is written out for all FIELD variables. Note: the requested point has to actually lie in the solution domain; if it does not, no output will result. Each point has an integer numeric tag, followed by a location in (x, y, z) space (even for two-dimensional solutions).

```
<HISTORY NUMBER=1>
  1 0.5 0.1 0.0
</HISTORY>
```

While mesh nodal locations may be scaled and shifted using tokens X_SCALE, X_SHIFT, Y_SCALE and Y_SHIFT (see e.g. § 2.5.1), the spatial locations declared in the HISTORY section should be given in the resulting mapped space.

2.6 Structure of a field file

Field files (with extensions .fld, .rst, .chk, .avg) have a common structure. They all commence with a 10-line (and 360-byte) ASCII header that briefly describes the file content, typically followed by sequenced binary storage of double-precision floating-point data with ordering that matches AuxField structure. It is possible to convert these data to human-readable ASCII format, via the convert utility — and back again, if desired, to binary format — but *Semtex* codes generally use the binary format for I/O of field files. Regardless of format, one should always be able to view the contents of the header using the Unix head command, which by default reads off the first 10 lines of a file.

For example, consider the structure of a two-dimensional field file; here we will use a 2D Taylor flow, considered again later in more detail in § 3.2.2. An example session file is supplied as mesh/taylor2; this has four elements, and each element has 11 collocation points on an edge (N_P=11), so a total of 121 solution data points. The solution is 2D2C, so the number of z-planes required is N_Z=1, but since this is a default value, it isn't specified in the session file. As in § 3.2.2, we will just use the compare utility to generate a restart file, and then look at its header.

```
$ compare taylor2 > taylor2.rst
$ head taylor2.rst
taylor2                    Session
Fri Aug 18 16:08:42 2023   Created
11   11   1    4           Nr, Ns, Nz, Elements
0                          Step
0                          Time
0.02                       Time step
0.01                       Kinvis
1                          Beta
uvp                        Fields written
binary IEEE little-endian  Format
```

As previously stated, this header is 360 bytes in length, including newline characters.[1] The third line of output tells us that there are 4 elements, 1 plane of data and each element is of size $N_r \times N_s = 11 \times 11 = 121$. (In *Semtex*, $N_r = N_s =$ N_P always.) The ninth line of output tells us that the file contains data for the velocity fields u, v, and pressure p. That means there should be $11 \times 11 \times 1 \times 4 \times 3 = 1452$ double-precision numbers in taylor2.rst. Each double-precision number consumes 8 bytes, thus we expect the size of taylor2.rst to be $1452 \times 8 + 360 = 11976$ bytes.

[1] If you carefully examine the header produced, you will find that the ninth, 'Fields written', is here padded at the end with 9 blank characters. This is to allow the number of fields to rise to a maximum of 49; if the number exceeds 25, the 'Fields written' string will be gradually pushed to the right, dropping characters in such a way that the total length of the line is still 49 characters. Why 49? Well, $49 = 2 \times 26 - 3$! The characters X, Y and Z are reserved.

```
$ wc -c < taylor2.rst
   11976
```

This example is simple as there is only a single plane of data for each field. Each plane contains the data for its elements, written in the order they are defined in the session file. (See e.g. figure 3.3, where the elements are labelled $1 \ldots 4$, and the locations of the data points are also indicated for our standard Gauss–Lobatto–Legendre element mesh structure.) The ordering of the data in each element is row-major, starting from the bottom-left corner and working across rows, then up, until the top-right corner is reached; this is also the ordering in the file.[2] Following the data for the first field (u), here a single plane of storage, we would find the data for the second field, then the third (and so on, if there were more fields indicated in the header). The extension to files with more planes of data is straightforward: the planes are written in order for the first field, then the second, etc.

Binary data in three-dimensional field files (where `N_Z` > 1) by default have physical-space ordering, even though most three-dimensional calculations in *Semtex* are carried out after a Fourier transformation in the z coordinate. If you wish to get the data to a Fourier-transformed state (say, to extract a particular two-dimensional Fourier mode), use the `transform` utility; see e.g. §§ 6.25 and 6.27. The ordering of data planes following real–complex transformation matches how they are stored in `AuxFields`: the 0th Fourier mode, which is the z-average at each (x, y) location and real, is stored as the first plane of data, while the `N_Z`$/2$th, 'Nyquist', or 'oddball' mode, also real, is stored as the second plane (typically, this will be set to zero since we do not evolve Nyquist modes). After these two planes of data we have the real and imaginary planes of each complex Fourier mode, in order, up to $(\texttt{N\_Z}/2) - 1$.

As stated above, field files can also be converted to pure ASCII form, for ease of human readability. This can be useful for checking purposes, but also allows manipulation of field files using Unix and Python text-processing utilities. See e.g. `utility/addquick`. In ASCII format, the ordering of data changes, such that the field variables are printed up in columns instead of one after another as in the binary case. Within each column, the ordering is as stated above for binary data: element-by-element and then plane-by-plane; see § 3.1.3 for an example of this structure. The utility `convert` does the conversion of formats, and can read and write from/to standard input and output (and so e.g. you may convert from one format to another and back using Unix pipes); see § 6.5.

Sometimes, field files will hold more than a single set of data for a complete field (e.g. a sequence of field dumps). In this case, each set of data commences with a new header (each with a different `Time`). To find out how many dumps are in a file, you could do something like:

```
$ convert taylor.rst | grep Session | wc -l
   1
```

2.7 Utilities

(See also the more extensive discourse of chapter 6.) Source code for these tools is found in the `utility` directory. Here is a summary of the most-used utilities:

`addfield`	Compute and add vorticity vector components, divergence, etc., to a field file.
`addquick`	An example shell script for adding arbitrary but simple fields of your choice using Unix `sed` and `awk` utilities together with `chop` and `convert`. (Quick *and dirty*!)
`calc`	A simple calculator that calls `femlib`'s function parser. The inbuilt parser functions and default `TOKENS` can be seen if you run `calc -h`.
`compare`	Generate restart files, compare solutions to a function.
`convert`	Convert field file formats (IEEE-big/little, to/from ASCII).
`eneq`	Compute terms in the energy transport equation.

[2] More generally, where the elements are distorted or rotated, from the first corner `NODE` along the edge to the second corner `NODE`, and on up to the third ('top-right corner') `NODE` location for any quadratic `ELEMENT` in a session file.

`assembly`	Generate global node numbering used in matrix assembly. The output of this utility is no longer directly used by simulation codes (since they now generate assembly mappings at runtime), but is supplied for checking purposes. It replaces the former utility called `enumerate`.
`integral`	Obtain the 2D integral of fields over the domain area.
`interp`	Interpolate a field file onto a (2D) set of points.
`meshplot`	Generate a PostScript plot of mesh from `meshpr` output.
`meshpr`	Generate 2D mesh locations for plotting or checking.
`noiz`	Add a Gaussian-distributed random perturbation to a field file.
`probe`	Probe a field file at a set of 2D/3D points. Different interfaces to `probe` are obtained through the names `probeline` and `probeplane`: make these soft links by hand.
`project`	Convert a field file to a different order interpolation.
`rectmesh`	Generate a template session file for a rectangular domain.
`rstress`	Postprocess to compute Reynolds stresses from a file of time-averaged variables.
`sem2tec`	Convert field files to AMTEC *Tecplot* format. Note that by default, `sem2tec` interpolates the original GLL-mesh-based data onto a (isoparametrically mapped) uniform mesh for improved visual appearance. Sometimes it is useful to see the original data (and mesh); for this use the `-n 0` command-line argument to `sem2tec`.
`sem2vtk`	Convert field files to VTK format (*VisIt, ParaView*).
`transform`	Take Fourier, Legendre, modal basis transform of a field file. Invertible.
`wallmesh`	Extract the mesh nodes corresponding to surfaces with the `wall` group.

Chapter 3

Example applications

We will run through some examples to illustrate input files, utility routines, and use of the solvers. While most users will likely be primarily interested in solving Navier–Stokes problems, we commence by outlining the use of the elliptic solver.

3.1 Elliptic equations

The elliptic solver can be used to solve Laplace, Poisson or Helmholtz problems in two-dimensional and three-dimensional Cartesian and cylindrical coordinate systems, vide (2.1). From a code development viewpoint it also provides a means to test new formulations of elliptic solution routines which are used in the Navier–Stokes type solvers; historically, `elliptic` was written and tested prior to `dns`. In this section we illustrate the use of the elliptic solver for a two-dimensional Laplace problem, $\nabla^2 c = 0$. In this case the function

$$c(x, y) = \sin(x) \exp(-y) \tag{3.1}$$

satisfies Laplace's equation and is used to set the boundary conditions. This example illustrates the methods used to set BCs and also to generate curved element boundaries. Also we will demonstrate the selection of the iterative PCG solver (Barrett et al.; 1994) as an alternative to the default direct Cholesky solver (Anderson et al.; 1999). The session file `laplace7` shown below is provided in the `mesh` subdirectory.

```
################################################################################
# Laplace problem on unit square, BC c(x, y) = sin(x)*exp(-y)
# is also the analytical solution.  Use essential (Dirichlet) BC
# on upper, curved edge, with natural (Neumann) BCs elsewhere.

<FIELDS>
  c
</FIELDS>

<USER>
        Exact sin(x)*exp(-y)
        c =   sin(x)*exp(-y)
</USER>

<TOKENS>
        N_P      = 11
        TOL_REL  = 1e-12
        STEP_MAX = 1000
</TOKENS>

<GROUPS NUMBER=4>
```

```
        1       d       value
        2       a       slope
        3       b       slope
        4       c       slope
</GROUPS>

<BCS NUMBER=4>
        1       d       1
                        <D>     c =  sin(x)*exp(-y)     </D>
        2       a       1
                        <N>     c = -cos(x)*exp(-y)     </N>
        3       b       1
                        <N>     c =  cos(x)*exp(-y)     </N>
        4       c       1
                        <N>     c =  sin(x)*exp(-y)     </N>
</BCS>

<NODES NUMBER=9>
        1       0.0     0.0     0.0
        2       0.5     0.0     0.0
        3       1.0     0.0     0.0
        4       0.0     0.5     0.0
        5       0.5     0.5     0.0
        6       1.0     0.5     0.0
        7       0.0     1.0     0.0
        8       0.5     1.0     0.0
        9       1.0     1.0     0.0
</NODES>

<ELEMENTS NUMBER=4>
        1       <Q>  1  2  5  4  </Q>
        2       <Q>  2  3  6  5  </Q>
        3       <Q>  4  5  8  7  </Q>
        4       <Q>  5  6  9  8  </Q>
</ELEMENTS>

<SURFACES NUMBER=8>
        1       1       1       <B>     c       </B>
        2       2       1       <B>     c       </B>
        3       2       2       <B>     b       </B>
        4       4       2       <B>     b       </B>
        5       4       3       <B>     d       </B>
        6       3       3       <B>     d       </B>
        7       3       4       <B>     a       </B>
        8       1       4       <B>     a       </B>
</SURFACES>

<CURVES NUMBER=1>
        1       4       3       <ARC>   1.0     </ARC>
</CURVES>
```

Refer to § 2.5 for a discussion of the purposes of each of the sections within this file. Within the TOKENS section, N_P=11 declares that there will be 11 points along the side of each element and hence, two-dimensional element shape functions are tensor products of 10th-order polynomials. TOL_REL=1e-12 directs the iterative preconditioned conjugate gradient solver, if used, to stop when the solution residual $||r^{(i)}|| = ||Ax^{(i)} - b|| < \texttt{TOL\_REL} \times ||b||$, while STEP_MAX=1000 directs that no more than 1000 iterations be taken in striving to reach TOL_REL; note that these values are only

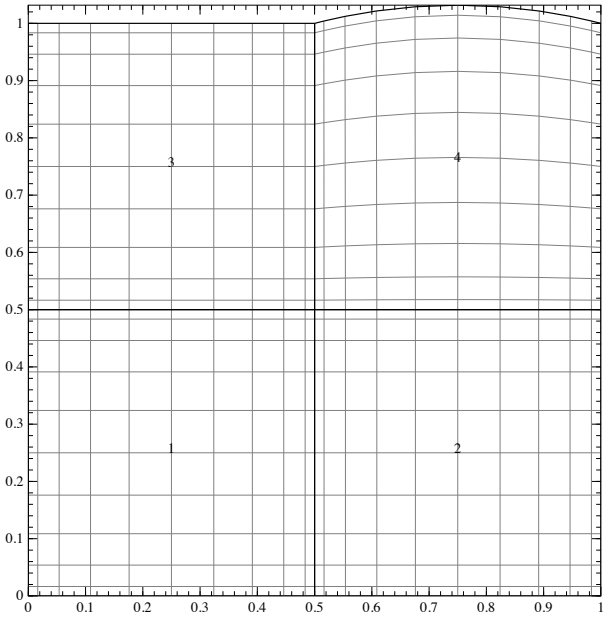


Figure 3.1: The mesh corresponding to the `laplace7` session file. Note that the number of points along the edge of every element is `N_P = 11` as declared in the `TOKENS` section and that internal points are placed inside non-rectangular elements, e.g. element 4, using isoparametric mapping. This image was produced using the `meshplot` utility.

relevant if an iterative solver is chosen instead of default direct solution, and that the default values of the two tokens are respectively `1e-8` and `500`, as may be established by running `calc -h`.

3.1.1 Curved element edges (and plotting the mesh)

The mesh for this problem consists of a set of four elements, one of which, the 3rd edge of element 4, specified in the `CURVES` section, is a a circular arc. Both `ARC` and `SPLINE` type curves are currently implemented. For the `ARC` type, the parameter supplies the radius of the curve: a positive value implies that the curve makes the element convex on that side, while a negative value implies a concave side. Note that where elements mate along a curved side, the curve must be defined twice, once for each element, and with radii of different signs on each side. The mesh for this problem can be seen in figure 3.1; mesh output is generated by running the `meshpr` utility, from which a PostScript file may be generated and visualised using the `meshplot` utility.

```
$ meshpr laplace7 | meshplot -i -n -o mesh.ps -d gv
```

If you have the gv utility installed (for X11 display of PostScript) you should see a plot like figure 3.1. Alternatively you could (again, assuming you have an appropriate utility installed) convert the output to PDF, e.g.

```
$ meshpr laplace7 | meshplot -i -n | ps2pdf - > laplace7.pdf
```

For the `SPLINE` type, the parameter supplies the name of an ASCII file which contains a list of (x, y) coordinate pairs (white-space delimited). Naturally, the list of points should be in arc-length order. A single file can be used to supply the curved edges for a set of element edges. The vertices of the relevant elements do not have to lie exactly on the splined curve — if they do not, those vertices get shifted to the intersection of the projection of the straight line joining the original vertex position and its neighbouring 'curve-normal' vertex, and the cubic spline joining the points in the file. On the other hand, it is good practice to ensure that the declared vertex locations lie close to the spline, and to visually check the mesh that is produced.

3.1.2 Boundary conditions

The sections GROUPS and BCS are used to specify boundary conditions for the problem. The GROUPS section associates a character group tag (e.g. d) with a string (e.g. value), but note that different groups can be associated with the same string[1]. Groups a, b and c will be used to set natural (i.e. slope or Neumann), boundary conditions ($\partial c/\partial n = \text{value}$), while group a will be used to impose an essential or Dirichlet condition ($c = \text{value}$).

The BCS section is used to define the boundary conditions which will be applied for each group. For each group, after the numeric tag (ignored) appears the character for that group, then the number of BCs that will be applied: this corresponds to the number of fields in the problem, in this case 1 (c). BCs are typically of Dirichlet, Neumann, or Robin/mixed type (note that domain periodicity can be employed but that this does not constitute a boundary condition). So in this case we will declare the BC types to be D (Dirichlet) for group d and N (Neumann) for groups a, b and c. On Neumann boundaries, the value which must be supplied is the slope of the solution along the outward normal to the solution domain. Note the fact that the BCs can be set using the function parser, using the built-in functions and variables, also any symbols defined in the TOKENS section, as well as the spatial variables x, y and z. The BCs can also be functions of time, t, and for time-varying problems, which this is not, the boundary conditions are re-evaluated every time step.

The BC groups are associated with element edges in the SURFACES section. Periodic edges, Dirichlet, Neumann and mixed boundary conditions can be arbitrarily combined in a problem. Dirichlet conditions over-ride Neumann ones where they meet (say at the corner node of an element). See further discussion on boundary conditions in § 3.7 below.

3.1.3 Running the codes

We will run the elliptic solver and compare the computed solution to the analytical solution. We will first run with the default direct solver:

```
$ elliptic laplace7
-- Forcing                 : set to zero
-- Initial condition       : set to zero
-- Installing matrices for field 'c' [*]
-- REMARK: Field: 'c' error norms (inf, L2, H1): 8.54872e-15  2.15768e-15  5.46269e-14
```

The 'Installing matrices' message indicates that a direct solution will be carried out. Next try the iterative (PCG) solver using the -i command-line option to elliptic:

```
$ elliptic -i laplace7
-- Forcing                 : set to zero
-- Initial condition       : set to zero
-- REMARK: Field: 'c' error norms (inf, L2, H1): 8.32201e-13  2.93711e-13  5.01449e-12
```

In this case the direct solver is more accurate, but comparable accuracy with the iterative solver could be obtained by decreasing TOL_REL in the TOKENS section (and if necessary by further increasing STEP_MAX).

For post-processing we can prepare a *Tecplot* input file (which can also be read by *VisIt* and *Paraview*)

```
$ meshpr laplace7 | sem2tec laplace7.fld
```

which produces laplace7.plt, an input file format which can be read by *Tecplot*. A contour plot of the solution obtained with *Tecplot* is shown in figure 3.2.

[1] This allows actions to be taken over a set of BCs which share the same string.

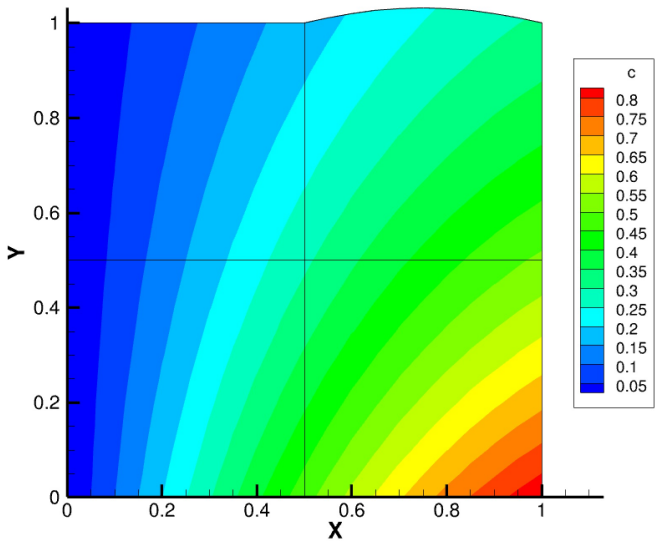


Figure 3.2: Solution corresponding to the `laplace7` session file. This image was produced using *Tecplot*.

3.1.4 Laplace, Poisson, Helmholtz problems

Poisson and Helmholtz problems differ from Laplace problems in that they have a forcing term. For elliptic problems this is supplied in the `USER` section by a line of the type `Forcing <string>` where `<string>` is ASCII text without white space which can be parsed, e.g. either a numeric constant or a function of spatial variables can be supplied. In the case of Helmholtz problems we need also to supply a positive numerical value for the token `LAMBDA2`, equivalent to the constant λ^2.

If one has an analytical solution to the problem in question, this can also be supplied in the `USER` section with a line of the type `Exact <string>` where again `<string>` is ASCII text which can be parsed. The outcomes are compared to the computed numerical solutions and various error norms are reported, as we saw in § 3.1.3. An analytical solution isn't required to run the solver, so this line is optional.

Various session files for elliptic problems can be found in the `mesh` subdirectory, with fairly obvious names. One of these, `laplace8` is for a three-dimensional problem in cylindrical coordinates. If a solution is desired in cylindrical coordinates, one must also set the token `CYLINDRICAL=1`; this convention applies in general for *Semtex* codes such as `dns` and the utility programs too. The default value of `CYLINDRICAL` is 0 (corresponding to Cartesian coordinates).

3.2 2D Taylor flow

Taylor flow is an analytical solution to the Navier–Stokes equations. In the x–y plane the solution is

$$u = -\cos(\pi x)\sin(\pi y)\exp(-2\pi^2\nu t), \tag{3.2}$$

$$v = +\sin(\pi x)\cos(\pi y)\exp(-2\pi^2\nu t), \tag{3.3}$$

$$p = -(\cos(2\pi x)+\cos(2\pi y))\exp(-4\pi^2\nu t)/4 + \text{const}. \tag{3.4}$$

The solution is doubly periodic in space, with periodic length 2. Since the solution is periodic in both directions, no boundary conditions are required but the periodicity must be specified in the `SURFACES` section. As usual for Navier–Stokes solutions, the pressure can only be specified up to an arbitrary constant. An interesting feature of this solution is that the nonlinear and pressure gradient terms balance one another, leaving a diffusive decay of the initial condition — this property is occasionally useful for checking codes.

3.2.1 Session file

Below is the complete input or *session* file we will use; it has four elements, each of the same size, with 11 nodes along each edge. This session file, `taylor2`, is supplied in the `mesh` subdirectory.

```
###############################################################################
# 2D Taylor flow in the x--y plane has the exact solution
#
#       u = -cos(PI*x)*sin(PI*y)*exp(-2.0*PI*PI*KINVIS*t)
#       v =  sin(PI*x)*cos(PI*y)*exp(-2.0*PI*PI*KINVIS*t)
#       w =  0
#       p = -0.25*(cos(2.0*PI*x)+cos(2.0*PI*y))*exp(-4.0*PI*PI*KINVIS*t)
#
# Use periodic boundaries (no BCs).

<USER>
        u = -cos(PI*x)*sin(PI*y)*exp(-2.0*PI*PI*KINVIS*t)
        v =  sin(PI*x)*cos(PI*y)*exp(-2.0*PI*PI*KINVIS*t)
        p = -0.25*(cos(TWOPI*x)+cos(TWOPI*y))*exp(-4.0*PI*PI*KINVIS*t)
</USER>

<FIELDS>
        u v p
</FIELDS>

<TOKENS>
        N_TIME  = 2
        N_P     = 11
        N_STEP  = 20
        D_T     = 0.02
        Re      = 100.0
        KINVIS  = 1.0/Re
        TOL_REL = 1e-12
</TOKENS>

<NODES NUMBER=9>
        1       0.0       0.0       0.0
        2       1.0       0.0       0.0
        3       2.0       0.0       0.0
        4       0.0       1.0       0.0
        5       1.0       1.0       0.0
        6       2.0       1.0       0.0
        7       0.0       2.0       0.0
        8       1.0       2.0       0.0
        9       2.0       2.0       0.0
</NODES>

<ELEMENTS NUMBER=4>
        1 <Q> 1 2 5 4 </Q>
        2 <Q> 2 3 6 5 </Q>
        3 <Q> 4 5 8 7 </Q>
        4 <Q> 5 6 9 8 </Q>
</ELEMENTS>

<SURFACES NUMBER=4>
        1       1       1       <P>     3       3       </P>
        2       2       1       <P>     4       3       </P>
        3       2       2       <P>     1       4       </P>
        4       4       2       <P>     3       4       </P>
```

</SURFACES>

(Below we give a brief reprise of some information we earlier dealt with in § 2.5.)

The first section of the file in this case contains comments; a line anywhere in the session file which starts with a # is considered to be a comment. Following that are a number of sections which are opened and closed with matching keywords in HTML style (e.g. <USER>–<\USER>). Keywords are not case sensitive. The complete list of keywords is: TOKENS, FIELDS, GROUPS, BCS, NODES, ELEMENTS, SURFACES, CURVES and USER. Depending on the problem being solved, some sections may not be needed, but the minimal set is: FIELDS, NODES, ELEMENTS and SURFACES. Anywhere there is likely to be a long list of inputs within the sections, the NUMBER of inputs is also required; this currently applies to GROUPS, BCS, NODES, ELEMENTS, SURFACES and CURVES. In each of these cases the numeric tag appears first for each input, which is free-format. The order in which the sections appear in the session file is irrelevant.

The USER section is ignored by the solvers, and is used instead by utilities — in this case it will be used by the compare utility both to generate the initial condition or *restart* file and to check the computed solution. This section declares the variables corresponding to the solution fields with the corresponding analytical solutions. The variables x, y, z and t can be used to represent the three spatial coordinates and time. Note that some constants such as PI and TWOPI are predefined, while others, like KINVIS, are set in the TOKENS section. Note also the use of predefined functions, accessed through an inbuilt function parser.[2]

The FIELDS section declares the one-character names of solution fields. The names are significant: u, v and w are the three velocity components (we only use u and v here for a 2D solution and the w component is always the direction of Fourier expansions), p is the pressure field. The field name c is also recognised as a scalar field for certain solvers, e.g. the elliptic solver.

In the TOKENS section, second-order accurate time integration is selected (N_TIME = 2) and the number of Lagrange knot points along the side of each element is set to 11 (N_P = 11), corresponding to the use of 10th-order polynomials, and giving two-dimensional elemental shape functions which are tensor-products of 10th-order Lagrange polynomials.[3] The code will integrate for 20 timesteps (N_STEP = 20) with a timestep of 0.02 (D_T = 0.002). The kinematic viscosity is set as the inverse of the Reynolds number (100): note the use of the function parser here. Finally the relative tolerance used as a stopping test in the PCG iteration used to solve the viscous substep on the first timestep is set as 1.0×10^{-12}.

The shape of the mesh is defined by the NODES and ELEMENTS sections. Here there are four elements, each obtained by connecting the corner nodes in a counterclockwise traverse. The x, y and z locations of the nodes are given, and the four numbers given for the nodes of each element are indices within the list of nodes.

In the final section (SURFACES), we describe how the edges of elements which define the boundary of the solution domain are dealt with. In this example, the solution domain is periodic and there are no boundary conditions to be applied, so the SURFACES section describes only periodic (P) connections between elements. For example, on the first line, side 1 of element 1 is declared to be periodic with side 3 of element 3 — side 1 runs between the first and second nodes, while side 3 runs between the third and fourth.

3.2.2 Running the codes

Assume we're in the dns directory of the distribution, that the compare, meshpr and sem2tec utilities have been compiled, as well as the dns simulation code.

```
$ cp ../mesh/taylor2 .
```

[2] The built-in functions and predefined constants can be found by running calc -h.

[3] The minimum accepted value of N_P = 2, corresponding to (bi)linear shape functions. The practicable upper value is around 20.

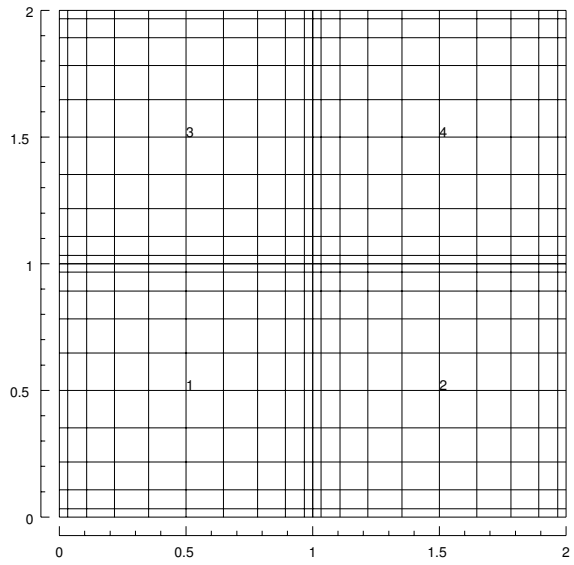


Figure 3.3: The mesh corresponding to the `taylor2` session file. This image was produced by using *SuperMongo* macros.

First we'll examine the mesh, in this case using *SuperMongo* macros supplied with the *Semtex* distribution as an alternative to using the `meshplot` utility.

```
$ meshpr taylor2 > taylor2.msh
$ sm
Hello Hugh, please give me a command
: meshplot taylor2.msh 1
Read lines 1 to 1 from taylor2.msh
Read lines 2 to 485 from taylor2.msh
: meshnum
: meshbox
: quit
```

You should have seen a plot like that in figure 3.3. (Note: while you are building up the mesh parts of a session file, perhaps by hand, you may use `meshpr -c` to suppress some of the checking for matching element edges and curved boundaries that `meshpr` does by default.)

The `compare` utility is used to generate a file of initial conditions using information in the `USER` section of a session file. This *restart* file contains binary data, but we'll have a look at the start of it by converting it to ASCII format. Also, the header of these files is always in ASCII format, and so can be examined directly using the Unix `head` command.

```
$ compare taylor2 > taylor2.rst
$ convert taylor2.rst | head -20
taylor2                   Session
Wed Aug 13 21:39:47 1997  Created
11   11   1    4          Nr, Ns, Nz, Elements
0                         Step
0                         Time
0.02                      Time step
0.01                      Kinvis
1                         Beta
uvp                       Fields written
ASCII                     Format
     0.000000000      0.000000000    -0.5000000000
```

```
     0.000000000     0.1034847104    -0.4946454574
     0.000000000     0.3321033052    -0.4448536974
     0.000000000     0.6310660897    -0.3008777952
     0.000000000     0.8940117093    -0.1003715318
     0.000000000      1.000000000      0.000000000
     0.000000000     0.8940117093    -0.1003715318
     0.000000000     0.6310660897    -0.3008777952
     0.000000000     0.3321033052    -0.4448536974
     0.000000000     0.1034847104    -0.4946454574
```

Then the `dns` solver is run to generate a solution or *field* file, `taylor2.fld`. This has the same format as the restart file.

```
$ dns taylor2
-- Initial condition       : read from file taylor2.rst
-- Coordinate system       : Cartesian
   Solution fields         : uvp
   Number of elements      : 4
   Number of planes        : 1
   Number of processors    : 1
   Polynomial order (np-1) : 10
   Time integration order  : 2
   Start time              : 0
   Finish time             : 0.4
   Time step               : 0.02
   Number of steps         : 20
   Dump interval (steps)   : 20 (checkpoint)
-- Installing matrices for field 'u' [*]
-- Installing matrices for field 'v' [.]
-- Installing matrices for field 'p' [*]
Step: 1  Time: 0.02
Step: 2  Time: 0.04
Step: 3  Time: 0.06
Step: 4  Time: 0.08
Step: 5  Time: 0.1
Step: 6  Time: 0.12
Step: 7  Time: 0.14
Step: 8  Time: 0.16
Step: 9  Time: 0.18
Step: 10  Time: 0.2
Step: 11  Time: 0.22
Step: 12  Time: 0.24
Step: 13  Time: 0.26
Step: 14  Time: 0.28
Step: 15  Time: 0.3
Step: 16  Time: 0.32
Step: 17  Time: 0.34
Step: 18  Time: 0.36
Step: 19  Time: 0.38
Step: 20  Time: 0.4
```

Note the difference in the information following the three '`Installing matrices`' messages: every instance of '`*`' indicates a new matrix system is being computed, while a '`.`' indicates that a previously computed system will be used: in this case, the matrix system for '`v`' can re-use the matrix system for '`u`', since the Helmholtz constant and boundary conditions match.

We can use `compare` to examine how close the solution is to the analytical solution. The output of `compare` in this case is a field file which contains the difference: since we're only interested in seeing error norms here, we'll discard this field file.

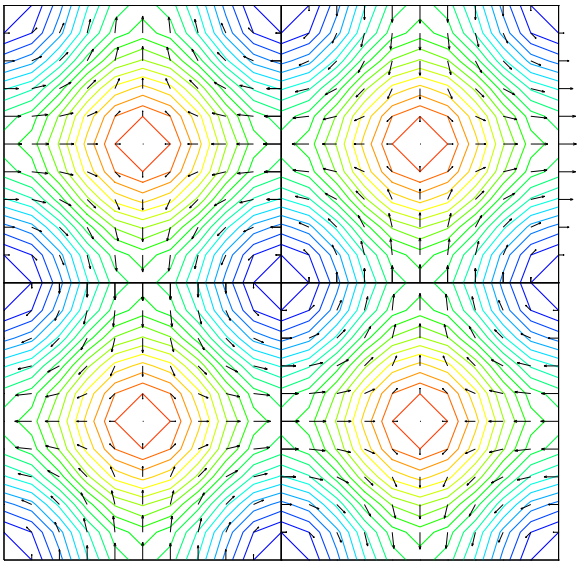


Figure 3.4: Solution to the `taylor2` problem, visualized using *Tecplot*.

```
$ compare taylor2 taylor2.fld > /dev/null
Field 'u': norm_inf: 1.127e-05
Field 'v': norm_inf: 1.127e-05
Field 'p': norm_inf: 4.224e-01
```

The velocity error norms are small, as expected, but the pressure norm will always be arbitrary, corresponding to the fact that the pressure can only be specified to within an arbitrary constant — hence the reported errors for pressure may be quite large, but this isn't a cause for alarm.

Finally we will use `sem2tec` to generate a *Tecplot* input file. The *Tecplot* utility `preplot` must also be in your `path`; source for this code is supplied in the `utility` subdirectory.

```
$ sem2tec -m taylor2.msh taylor2.fld
```

This produces `taylor2.plt` which can be used as input to *Tecplot*. The plot in figure 3.4 was generated using *Tecplot* and shows pressure contours and velocity vectors. Notice that largely for cosmetic reasons, by default `sem2tec` interpolates the results from the Gauss–Lobatto–Legendre grid (seen in figure 3.3) used in the computation to a uniformly-spaced grid of the same order (use `-n 0` to defeat this feature and see the actual solution values and on the original mesh).

3.3 3D Kovasznay flow

Here we will solve another viscous flow for which an analytical solution exists, the Kovasznay flow (we will call the session file `kovas3`). In the (x, y) plane, this flow is

$$u = 1 - \exp(\lambda x)\cos(2\pi y) \tag{3.5}$$
$$v = \lambda/(2\pi)\exp(\lambda x)\sin(2\pi y) \tag{3.6}$$
$$w = 0 \tag{3.7}$$
$$p = (1 - \exp(\lambda x))/2 + \text{const.} \tag{3.8}$$

where $\lambda = Re/2 - (0.25Re^2 + 4\pi^2)^{1/2}$.

Although the solution has only two velocity components, we will set up and solve the problem in three dimensions, with a periodic length in the z direction of 1.0 and eight z-planes of data. The

length in the z direction is set within the code by the variable BETA where $\beta = 2\pi/L_z$. The default value of BETA is 1, so we reset this in the TOKENS section using the function parser. The exact velocity boundary conditions are supplied on at the left and right edges of the domain, and periodic boundaries are used on the upper and lower edges (the domain has $-0.5 \leq y \leq 0.5$). Since the flow evolves to a steady state, first order timestepping is employed (N_TIME = 1).

The session file below is provided as mesh/kovas3.

```
###############################################################################
# Kovasznay flow in the x--y plane has the exact solution
#
#       u = 1 - exp(lambda*x)*cos(2*PI*y)
#       v = lambda/(2*PI)*exp(lambda*x)*sin(2*PI*y)
#       w = 0
#       p = (1 - exp(lambda*x))/2
#
# where lambda = Re/2 - sqrt(0.25*Re*Re + 4*PI*PI).
#
# This 3D version uses symmetry planes on the upper and lower boundaries
# with flow in the x-y plane.
#
# Solution accuracy is independent of N_Z since all flow is in the x--y plane.

<USER>
        u = 1.0-exp(LAMBDA*x)*cos(TWOPI*y)
        v = LAMBDA/(TWOPI)*exp(LAMBDA*x)*sin(TWOPI*y)
        w = 0.0
        p = 0.5*(1.0-exp(LAMBDA*x))
</USER>

<FIELDS>
        u v w p
</FIELDS>

<TOKENS>
        N_Z    = 8
        N_TIME = 1
        N_P    = 8
        N_STEP = 500
        D_T    = 0.008
        Re     = 40.0
        KINVIS = 1.0/Re
        LAMBDA = Re/2.0-sqrt(0.25*Re*Re+4.0*PI*PI)
        Lz     = 1.0
        BETA   = TWOPI/Lz
</TOKENS>

<GROUPS NUMBER=1>
        1       v       velocity
</GROUPS>

<BCS NUMBER=1>
        1       v       4
                        <D> u = 1-exp(LAMBDA*x)*cos(2*PI*y)             </D>
                        <D> v = LAMBDA/(2*PI)*exp(LAMBDA*x)*sin(2*PI*y) </D>
                        <D> w = 0.0                                     </D>
                        <H> p                                           </H>
</BCS>
```

```
<NODES NUMBER=9>
        1      -0.5    -0.5     0.0
        2       0      -0.5     0.0
        3       1      -0.5     0.0
        4      -0.5     0       0.0
        5       0       0       0.0
        6       1       0       0.0
        7      -0.5     0.5     0.0
        8       0       0.5     0.0
        9       1       0.5     0.0
</NODES>

<ELEMENTS NUMBER=4>
        1 <Q> 1 2 5 4 </Q>
        2 <Q> 2 3 6 5 </Q>
        3 <Q> 4 5 8 7 </Q>
        4 <Q> 5 6 9 8 </Q>
</ELEMENTS>

<SURFACES NUMBER=6>
        1       1       1       <P>     3       3       </P>
        2       2       1       <P>     4       3       </P>
        3       2       2       <B>     v               </B>
        4       4       2       <B>     v               </B>
        5       3       4       <B>     v               </B>
        6       1       4       <B>     v               </B>
</SURFACES>
```

3.3.1 'High-order' pressure boundary condition

Note that there is only one boundary group, and four boundary conditions must be set, corresponding to the four fields `u`, `v`, `w` and `p`. A new feature is a pressure BC of type `H`, which is an internally-computed Neumann boundary condition, (a High-order pressure BC) as described in Karniadakis et al. (1991). This is the kind of pressure BC that we would typically supply on all surfaces except on outflow boundaries or, in cylindrical coordinates, on the x-axis. The pressure BC is computed internally, so no value is required (if given, it will be ignored).

3.3.2 Running the codes

Since we already have an analytical solution for the problem, we may as well use that to generate an initial condition and then run for a moderate number of time steps (e.g. 200) to see if the discrete solution diverges much from this.

```
$ compare kovas3 > kovas3.rst
$ dns kovas3
```

After running `dns`, we confirm there is only a single dump in the field file `kovas3.fld`,[4] then run compare in order to examine the error norms for the solution.

```
$ convert kovas3.fld | grep -i session
kovas3                    Session
$ compare kovas3 kovas3.fld > /dev/null
Field 'u': norm_inf: 5.773e-05
Field 'v': norm_inf: 3.145e-05
```

[4] Note the use of `convert` and `grep` to see how many dumps are contained within a field file. More generally (e.g. within a script) we could pipe the outcome into `wc -l` to provide a numeric value.

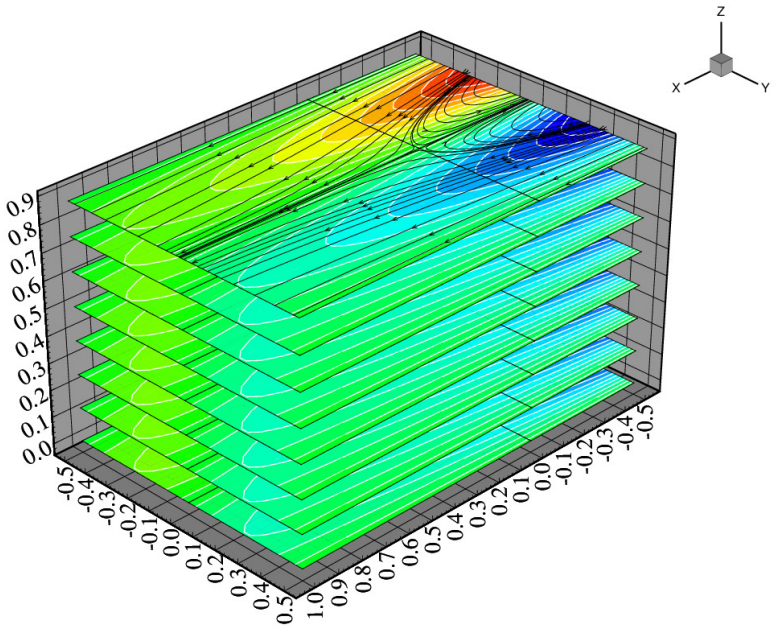


Figure 3.5: Solution to the `kovas3` problem, visualised using *Tecplot*. The plot shows contours of v velocity component and streamlines. Since the solution is in fact two-dimensional but we carried out a three-dimensional solution, the results are identical on each z-plane of data.

```
Field 'w': norm_inf: 0.000e+00
Field 'p': norm_inf: 9.350e-01
$ meshpr kovas3 | sem2tec -n20 kovas3.fld
```

The errors incurred are relatively small and could be reduced by re-running with increased polynomial order; in `kovas3`, `N_P = 8`, i.e. the polynomial order in each direction for each element is 7. Increasing `N_P` to 13 will reduce the errors in velocity by more than three orders of magnitude (check for yourself), illustrating the exponential convergence property of spectral element methods.

Finally we prepare input for *Tecplot*, projecting the interpolation to a 20×20 grid in each element in order to produce smoother contours. A view of the result can be seen in figure 3.5.

3.3.3 Valid values of `N_Z` in Semtex

This was a 'fake' three-dimensional problem since while we used `N_Z=8`, in fact the solution is two-dimensional, at least in Cartesian coordinates; the solution is invariant in the z-direction (in chapter 7 we will examine turbulent DNS which *is* three-dimensional). However, in passing, we note here the values which `N_Z` can take for three-dimensional solutions: since we use a 2–3–5 prime-factor complex–complex FFT (Temperton; 1992) together with a real–complex conversion (e.g. § 12.3, Press et al.; 1992) to carry out Fourier transforms, `N_Z` must be even (factorisable by 2) but can also have factors of 3 and 5. Hence valid values of `N_Z` are: $2, 4, 6, 8, 10, 12, 16, 20, 24, 30, 32 \ldots$. Of course `N_Z=1` is also valid but the solution will perforce be two-dimensional. However, owing to the use of complex–real conversion in which the Nyquist data are always set to zero, in fact a solution with `N_Z=2` will also always be two-dimensional (the same on both z-planes). Hence the minimum value of `N_Z` which in fact allows any three-dimensionality in *Semtex* is 4. For Navier–Stokes problems, flows which have `N_Z` > 1 must also have three velocity components.

3.4 Vortex breakdown — a cylindrical-coordinate problem

Here we will examine a problem which uses the cylindrical coordinate option of `dns`. The physical situation is a cylindrical cavity, $H/R = 2.5$ with the flow driven by a spinning lid at one end. At the Reynolds number we'll use, $Re = \Omega R^2/\nu = 2119$, a vortex breakdown is known to occur. The flow in this case is invariant in the azimuthal direction, but has three velocity components (it is 2D3C) and hence we use `N_Z=1`. In the cylindrical code, the order of spatial directions and velocity components is x, r, θ, though we retain the coordinate names x, y, z (i.e. $y \equiv r$ and $z \equiv \theta$, $v \equiv \boldsymbol{u}_r$, $w \equiv \boldsymbol{u}_\theta$).

Note that for a full circle in the azimuthal direction, `BETA = 1.0`, which is the default value. In fact, the value would not be used in the present solution, since all derivatives in the azimuthal direction are implicitly zero when `N_Z=1`. (But see § 3.4.1 below.) The session file below is provided as `mesh/vb1`.

```
################################################################################
# 15 element driven cavity flow.  2D/3C

<FIELDS>
        u v w p
</FIELDS>

<TOKENS>
        CYLINDRICAL = 1
        N_Z         = 1
        BETA        = 1.0
        N_TIME      = 2
        N_P         = 11
        N_STEP      = 100000
        D_T         = 0.01
        Re          = 2119
        KINVIS      = 1/Re
        OMEGA       = 1.0
        TOL_REL     = 1e-12
</TOKENS>

<GROUPS NUMBER=3>
        1       v       velocity
        2       w       wall
        3       a       axis
</GROUPS>

<BCS NUMBER=3>
        1       v       4
                        <D>     u = 0           </D>
                        <D>     v = 0           </D>
                        <D>     w = OMEGA*y     </D>
                        <H>     p               </H>
        2       w       4
                        <D>     u = 0           </D>
                        <D>     v = 0           </D>
                        <D>     w = 0           </D>
                        <H>     p               </H>
        3       a       4
                        <A>     u               </A>
                        <A>     v               </A>
                        <A>     w               </A>
                        <A>     p               </A>
</BCS>
```

```
<NODES NUMBER=24>
        1       0       0       0
        2       0.4     0       0
        3       0.8     0       0
        4       1.5     0       0
        5       2.4     0       0
        6       2.5     0       0
        7       0       0.15    0
        8       0.4     0.15    0
        9       0.8     0.15    0
        10      1.5     0.15    0
        11      2.4     0.15    0
        12      2.5     0.15    0
        13      0       0.75    0
        14      0.4     0.75    0
        15      0.8     0.75    0
        16      1.5     0.818   0
        17      2.4     0.9     0
        18      2.5     0.9     0
        19      0       1       0
        20      0.4     1       0
        21      0.8     1       0
        22      1.5     1       0
        23      2.4     1       0
        24      2.5     1       0
</NODES>

<ELEMENTS NUMBER=15>
        1       <Q>      1  2  8  7     </Q>
        2       <Q>      2  3  9  8     </Q>
        3       <Q>      3  4 10  9     </Q>
        4       <Q>      4  5 11 10     </Q>
        5       <Q>      5  6 12 11     </Q>
        6       <Q>      7  8 14 13     </Q>
        7       <Q>      8  9 15 14     </Q>
        8       <Q>      9 10 16 15     </Q>
        9       <Q>     10 11 17 16     </Q>
        10      <Q>     11 12 18 17     </Q>
        11      <Q>     13 14 20 19     </Q>
        12      <Q>     14 15 21 20     </Q>
        13      <Q>     15 16 22 21     </Q>
        14      <Q>     16 17 23 22     </Q>
        15      <Q>     17 18 24 23     </Q>
</ELEMENTS>

<SURFACES NUMBER=16>
        1       1       1       <B>     a       </B>
        2       2       1       <B>     a       </B>
        3       3       1       <B>     a       </B>
        4       4       1       <B>     a       </B>
        5       5       1       <B>     a       </B>
        6       5       2       <B>     v       </B>
        7       10      2       <B>     v       </B>
        8       15      2       <B>     v       </B>
        9       15      3       <B>     w       </B>
        10      14      3       <B>     w       </B>
        11      13      3       <B>     w       </B>
```

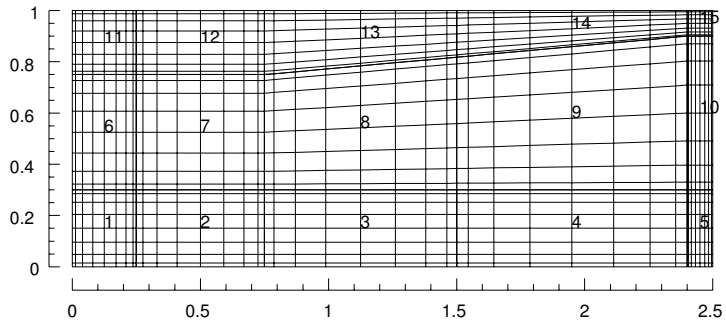


Figure 3.6: Mesh for the vortex breakdown problem. The spinning lid is at right. This image was produced using *SuperMongo* macros.

```
        12      12      3       <B>     w       </B>
        13      11      3       <B>     w       </B>
        14      11      4       <B>     w       </B>
        15      6       4       <B>     w       </B>
        16      1       4       <B>     w       </B>
</SURFACES>
```

The mesh for the problem is shown in figure 3.6, and the velocity field estimate obtained by running `dns` is shown compared to an experimental streakline flow visualisation on the front cover of this document.

3.4.1 BCs for cylindrical coordinates

A new feature here is the use of BCs of type `A` on the axis of the flow. Internally, the code sets the BC there either as zero essential or zero natural, depending on the physical variable and the Fourier mode. Owing to the coupling scheme used in the code (Blackburn and Sherwin; 2004), the boundary conditions for the radial and azimuthal velocities `v` and `w` must be of the same type within each group. A further restriction is that the group to which the axis belongs must have name `axis`.

Finally, say you wish to solve a cylindrical-coordinate problem where you know that, while the solution may be three-dimensional, there is an n-fold azimuthal symmetry (say $n = 3$). In that case, it is much cheaper to solve with `BETA=3`, and use one-third the number of azimuthal planes that would be required for `BETA=1`. One does not have to change the specification of axis boundary conditions when making such a change, as they are internally computed according to Fourier mode index and `BETA`.

3.5 Buoyancy driven flow in a cavity

Next we will examine a Navier–Stokes problem with an advected scalar (temperature) and Boussinesq buoyancy (see § 4.13.9) driving flow in a rectangular cavity. The session file shown below is provided as `tdrivcav1`.

```
# Thermal driven cavity problem for buoyancy-driven flow.
# Left edge is hot, right edge is cool, top and bottom are insulated.

<FIELDS>
        u v c p
</FIELDS>

<USER>
```

```
        u = 0.0
        v = 0.0
        c = T_MAX-x
        p = 0.0
</USER>

<FORCE>
        BOUSSINESQ_TREF    = 0.0
        BOUSSINESQ_BETAT   = RAYLEIGH*PRANDTL
        BOUSSINESQ_GRAVITY = 1.0
        BOUSSINESQ_GY      = -1.0
</FORCE>

<TOKENS>
        N_TIME   = 1
        N_P      = 10
        N_STEP   = 5000
        IO_FLD   = 1000
        D_T      = 0.0008
        T_MAX    = 1.0
        T_MIN    = 0.0
        PRANDTL  = 0.71
        RAYLEIGH = 1.0e4
        KINVIS   = PRANDTL
</TOKENS>

<GROUPS NUMBER=3>
        1       h       hot
        2       c       cold
        3       i       insulated
</GROUPS>

<BCS NUMBER=3>
        1       h       4
                        <D>     u = 0.0         </D>
                        <D>     v = 0.0         </D>
                        <D>     c = T_MAX       </D>
                        <H>     p               </H>
        2       c       4
                        <D>     u = 0.0         </D>
                        <D>     v = 0.0         </D>
                        <D>     c = T_MIN       </D>
                        <H>     p               </H>
        3       i       4
                        <D>     u = 0.0         </D>
                        <D>     v = 0.0         </D>
                        <N>     c = 0.0         </N>
                        <H>     p               </H>
</BCS>

<NODES NUMBER=9>
        1       0.0     0.0     0.0
        2       0.5     0.0     0.0
        3       1.0     0.0     0.0
        4       0.0     0.5     0.0
        5       0.5     0.5     0.0
        6       1.0     0.5     0.0
        7       0.0     1.0     0.0
```

```
        8       0.5     1.0     0.0
        9       1.0     1.0     0.0
</NODES>

<ELEMENTS NUMBER=4>
        1       <Q>     1 2 5 4         </Q>
        2       <Q>     2 3 6 5         </Q>
        3       <Q>     4 5 8 7         </Q>
        4       <Q>     5 6 9 8         </Q>
</ELEMENTS>

<SURFACES NUMBER=8>
        1       1       1       <B>     i       </B>
        2       2       1       <B>     i       </B>
        3       2       2       <B>     c       </B>
        4       4       2       <B>     c       </B>
        5       4       3       <B>     i       </B>
        6       3       3       <B>     i       </B>
        7       3       4       <B>     h       </B>
        8       1       4       <B>     h       </B>
</SURFACES>

$ dns tdrivcav1
-- Initial condition       : set to zero
-- Coordinate system       : Cartesian
   Solution fields         : uvcp
   Number of elements      : 4
   Number of planes        : 1
   Number of processors    : 1
   Polynomial order (np-1) : 9
   Time integration order  : 1
   Start time              : 0
   Finish time             : 4
   Time step               : 0.0008
   Number of steps         : 5000
   Dump interval (steps)   : 1000 (checkpoint)
-- Installing matrices for field 'u' [*]
-- Installing matrices for field 'v' [.]
-- Installing matrices for field 'c' [*]
-- Installing matrices for field 'p' [*]
Step: 1  Time: 0.0008
Step: 2  Time: 0.0016
Step: 3  Time: 0.0024
Step: 4  Time: 0.0032
Step: 5  Time: 0.004
...
...
Step: 4996  Time: 3.9968
Step: 4997  Time: 3.9976
Step: 4998  Time: 3.9984
Step: 4999  Time: 3.9992
Step: 5000  Time: 4
# CFL: 1.4, dt (max): 0.000572, dt (set): 0.0008 (139%), field: v, elmt: 1
# Divergence Energy:    0.000283
```

By now the structure of the session file should be fairly familiar, except perhaps for the specification of a thermally driven buoyancy body force in the FORCE section, for which you are referred ahead

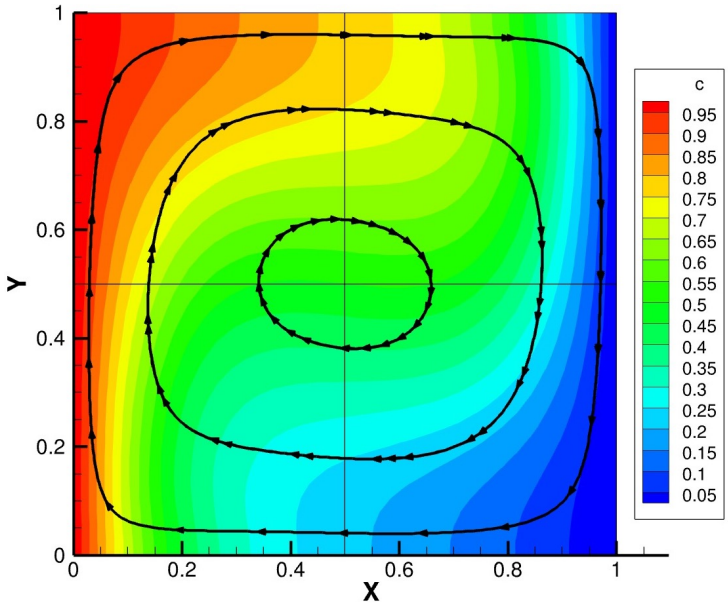


Figure 3.7: Solution for the buoyancy driven flow in a cavity: colour contours of temperature overlaid with streamlines. Compare results for $Ra = 1 \times 10^4$ in de Vahl Davis (1983). This image was produced using *Tecplot*.

to read § 4.13 and § 4.13.9 in particular. Note how variables are set to achieve a desired Rayleigh number $Ra = \mathrm{g}\beta_T L^3 (T_{\max} - T_{\min})/(\nu\alpha T_{\mathrm{ref}}) = 1 \times 10^4$.

3.6 Timestepping stability: CFL and divergence energy

We have not yet touched upon the `dns` output information regarding CFL (Courant–Friedrichs–Lewy) limits and divergence (as listed in the example above). These are reported in `dns` output every `IO_CFL` steps. The CFL estimates are computed according to methods outlined in § 6.3 of Karniadakis and Sherwin (2005) based on velocity component values and mesh sizes, and are roughly proportional to $u\Delta t/\Delta x$. Typically, we need to keep CFL values below around unity to maintain stability with explicit time integration methods of the kind used here for advection based on nonlinear terms. We see that at the end of integration, our CFL value is actually larger than this (1.4), though stable integration is achieved (partly since, as we are integrating towards a steady-state solution, we have used first-order timestepping rather than the second-order default). Also, we see which velocity component (v) is the most critical with respect to CFL instability, and in which element (1) this occurs.

On the lines following CFL estimate reports, we get another indication of solution robustness, the (average) divergence energy, which is $(2A)^{-1} \int (\boldsymbol{\nabla} \cdot \boldsymbol{u})^2 \, \mathrm{d}\Omega$ where $A = \int \mathrm{d}\Omega$ is the area of the domain. Our numerical method, which uses a time-splitting approach to integrating the Navier–Stokes equations, commits a divergence error that is asymptotically proportional to $|\Delta t|^{\texttt{N\_TIME}}$ (even when the mesh is adequate for spatial resolution). If the timestep is too large, we will also get large divergence energy: for solutions with length and velocity scales of order unity, a rule of thumb is that reported divergence energies should be well below unity in order for results to be reasonably well resolved in time. Typically, if the timestep is too large, CFL instability will set in and both the reported CFL and divergence energy values will start to become very large before the simulation terminates with a floating-point overflow error (Unix `SIG8`).

3.7 Boundary condition roundup

The basic types of boundary conditions the code can deal with are Dirichlet type (value of variable is set, a.k.a. an *essential* boundary condition in the finite element community) and Neumann type (boundary-normal gradient of value is approximated, a.k.a. a *natural* type boundary condition in the finite-element community). The code can also deal with boundary conditions of mixed type (a linear combination of Dirichlet and Neumann). We note that periodic domain boundary surfaces (<P>) are allowed, but strictly speaking, periodicity does not constitute a boundary condition as such. Also we remark that for our code(s), boundary conditions may only be set for variables involved in elliptic sub-problems where, owing to the MWR treatment, Neumann boundary conditions are implemented as integral approximations (which converge to the given value pointwise as resolution is increased), while Dirichlet conditions are 'lifted' out of the problem and imposed exactly to the values which are set by the user.

Standard Dirichlet and Neumann boundary conditions may be supplied as a string that can be parsed by the solver to obtain a real value, based on predefined and user-declared TOKENS, and also the space/time variables x, y, z and t. These strings are re-parsed at each time step, so time-varying boundary conditions are allowed. We have already seen examples of these BC declarations in §§ 3.1, 3.3 and 3.4. Note that the strings involved should not contain white space.

We have not yet described mixed boundary conditions. These are of type $\partial c/\partial n + K(c - C) = 0$ where C and K are constants. Here, n signifies the unit outward normal direction: $\partial c/\partial n = \boldsymbol{n} \cdot \boldsymbol{\nabla} \boldsymbol{c}$. Mixed boundary conditions are specified in the form <M> field = mulval;refval </M> where mulval is (a string that evaluates to) the real value K and refval is (a string that evaluates to) the real value C. At present for mixed BCs, unlike Dirichlet and Neumann boundary conditions, C and K are fixed at the values they initially evaluate to (not time-varying).

In Navier–Stokes type problems, various of the above boundary conditions for the velocity and pressure variables are typically combined in set ways. In some cases, the user does not provide values or choose the combination since the boundary conditions are computed internally. Below we supply as examples various typical boundary condition sets which would be located in the BCS section of a session file. As written, they are for two-component Navier–Stokes problems but the generalisation to three-component problems should be obvious.

See also § 3.2 of the *Dog* user guide for a discussion of sets of boundary conditions appropriate for symmetry and anti-symmetry boundaries.

3.7.1 No-slip wall

```
<D> u = 0.0 </D>
<D> v = 0.0 </D>
<H> p       </H>
```

The tag H for pressure (p) denotes an internally computed 'high-order' Neumann condition, as originally described by Karniadakis et al. (1991). If the associated GROUP string is wall then tractions will contribute to the integrated values found in session.flx file, see § 4.5.

3.7.2 Inflow or prescribed-velocity boundary

```
<D> u = 1.0 </D>
<D> v = 0.0 </D>
<H> p       </H>
```

Note that either of the supplied values can be a string to be evaluated by the parser at each timestep.

3.7.3 Slip (no-penetration) boundary

```
<N> u = 0.0 </N>
<D> v = 0.0 </D>
<H> p       </H>
```

Note that in this case, the boundary needs to be aligned with the x axis. At present there is no way to set a slip boundary which is inclined or curved; it must be parallel to either the x or y axis.

3.7.4 'Stress-free' outflow boundary

```
<N> u = 0.0 </N>
<N> v = 0.0 </N>
<D> p = 0.0 </D>
```

This is a restricted approximation to a true stress-free boundary where the tractions are zero. This boundary is also stress-free but achieves the condition by ensuring that the viscous and pressure tractions are individually zero, rather than their sum.

3.7.5 Energy-stable open boundary

```
<O> u </O>
<O> v </O>
<O> p </O>
```

This is a set of computed boundary conditions (computed Robin/mixed for velocity and pressure). These boundary conditions were originally described in Dong (2015) (see equations 37 and 38 there), and are based on maintaining boundedness of kinetic energy within the domain. It is an excellent combination for maintaining stability for flows in short open domains, where the use of the 'stress-free' condition described in § 3.7.4 can lead to catastrophe if significant inflow occurs over an outflow boundary, or for allowing inflows on ingestion boundaries (such as outboard of a jet issuing from a wall). Type `O` boundaries must set the string `open` in their associated `GROUP`. In addition one may set the tokens `DONG_UODELTA` and `DONG_DO` (see Dong; 2015, these are $U_o\delta$ and D_o), with default values 0.05 and 1.0 respectively. For a scalar variable (c), a zero Neumann condition is set on these boundaries.

3.7.6 Axis boundary

```
<A> u </A>
<A> v </A>
<A> p </A>
```

This is a set of Fourier-mode dependent homogeneous Dirichlet and Neumann boundary conditions to be used when the boundary coincides with the x axis of a cylindrical coordinate system as described in Blackburn and Sherwin (2004). Type `A` boundaries must set the string `axis` in their associated `GROUP`.

3.8 Fixing problems

You are liable to come up against a few generic problems when making and running your own cases. Here we will restrict discussion to Navier–Stokes problems and `dns`. The code will output an estimate of the CFL-timestep every `IO_CFL` timesteps (default 50), along with the average divergence of the solution (in the operator-splitting used, incompressibility is only ensured in the spatial-convergence limit). The CFL estimate is generally quite reliable and provides guidance as to which velocity

component and element number is the most critical, but also the divergence energy provides an excellent diagnostic of trouble! If velocity and length scales are of order unity, the reported divergence energy should be much less than unity; if the divergence is comparatively large then either the solution is blowing up,[5] or the spatial resolution is inadequate, or both.

By far the most common problem is that the solution will have a CFL-type instability brought about by using too large a time-step; this instability is unavoidably associated with using explicit time integration for the advection terms in the Navier–Stokes equations. This problem is easily enough fixed: try reducing `D_T`, and if required, increasing `N_STEP` to maintain the same integration interval. Obviously you will typically want `D_T` as large as possible, so if the problem runs stably, increase the timestep as much as is reasonable. If the velocity and timescales are of order unity, then the maximum timestep would typically be of order two orders of magnitude smaller (`0.01`). Note that CFL-stability will decrease with increasing time-integration order (`N_TIME`).

If the solution persists in blowing up when the timestep is reduced, the next most common cause is that there is inflow across an outflow boundary (in which case the problem is ill-posed, however, in practice *some* inflow across an outflow boundary over restricted times may be present without causing difficulty). To check if this is the cause, you could put some history points near the outflow (see § 4.8), but the best method of diagnosis is to run the solution up to a time when divergence starts to increase markedly, then use *Tecplot* or some other postprocessor to examine the solution near the outflow. This problem has been largely circumvented in *Semtex* V8 using the open BC set described by Dong (2015), see § 3.7.5 above, though it cannot overcome all problems. In pathological cases, fixing the problem will generally require the mesh to be altered: sometimes the mesh is badly structured near the outflow (e.g. element sizes have been varied too rapidly); sometimes the problem can be overcome by extending the domain downstream; sometimes the domain needs to be reshaped (e.g. by contracting it in the cross-flow direction) so that there will be no outflow over the inflow boundary. If all else fails, consider the methods of § 4.13.4 to force the velocity near the outflow to be something more computationally tractable (if unphysical).

Finally: if you can't remember the syntax of a particular *Semtex* command, most of the executables will issue a usage prompt when requested with the `-h` command-line flag, along the lines of regular Unix commands. For example:

```
$ dns -h
Usage: dns [options] session-file
  [options]:
  -h       ... print this message
  -f       ... freeze velocity field (scalar advection/diffusion only)
  -i       ... use iterative solver for viscous steps
  -v[v...] ... increase verbosity level
  -chk     ... turn off checkpoint field dumps [default: selected]
  -S|C|N   ... regular skew-symm || convective || Stokes advection
```

If that is not enough help then you might consider examining at least the header of associated source files, especially of the utilities.

3.9 Execution speed

3.9.1 Serial (per processor) speed

Semtex makes good use of the BLAS, so it can be worth seeking fast implementations. Especially, performance of the matrix–matrix multiplication routine `dgemm` is critical. Many math libraries, e.g. `openblas` and vendor-supplied math libraries such as AMD's `acml`, Intel's `mkl` or Apple's `Accelerate` framework, now include some version of Kazushige Goto's fast implementation of `dgemm`.

[5] One wag suggested the name *Semtex* was associated with this property of the solutions.

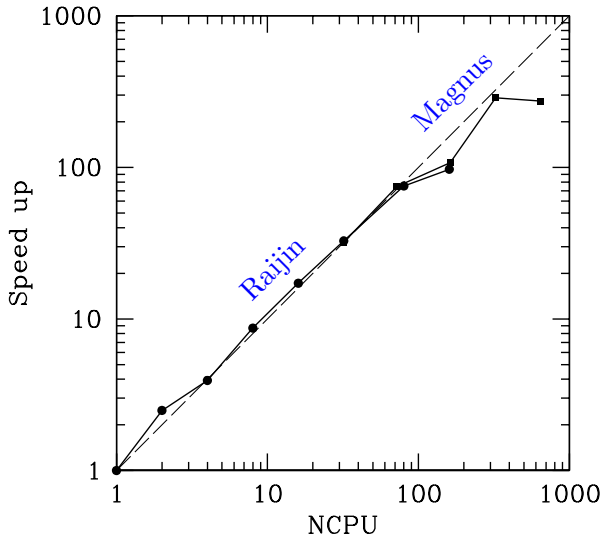


Figure 3.8: Hard scaling (speed-up for problem of a fixed size) relations for Fourier-parallel DNS of a turbulent pipe flow obtained at two different supercomputer facilities using MPI. The speed-up is approximately linear with number of processes until communications overheads become dominant (problem-dependent). (Reproduced from Blackburn et al.; 2019).

As Reynolds numbers increase (i.e. `KINVIS` decreases), the viscous Helmholtz matrices in the operator splitting become more diagonally dominant and better conditioned. In this case, you may find that iterative (PCG) solution of the viscous step (obtained by setting `ITERATIVE=1` or running `dns -i`) is actually faster than the direct solution that is obtained by default. This is nice because additionally, less memory is required. It is generally worth checking this if you plan an extended series of runs, and Reynolds numbers are large.

3.9.2 Parallel execution

At present, the two primary solvers in *Semtex* support parallel MPI-based execution for three-dimensional problems. The associated solvers are called `elliptic_mp` (which gets little use — it mainly exists for completeness and testing purposes) and `dns_mp`. If an MPI system (e.g. `openmpi`, `mpich`, `lam/mpi`) is present on your machine, the `cmake` build system should detect that and compile these two extra executables.

For three-dimensional problems (`N_Z` ≥ 4, see §3.3.3), the solution can be made parallel across (two-dimensional) Fourier modes. No change to the problem's `session` file is required; one just needs to change the executable from `dns` to `dns_mp` and pass execution over to `mpirun` (or local equivalent) for administration. The maximum number of processors which can be employed is `N_Z` $/2$, because each two-dimensional Fourier mode is complex (has real and imaginary parts), must be even, and must be congruent with the 2-3-5 prime-factor FFT used. (Don't be too concerned: if you choose an inappropriate value, an error message will be issued and execution will be terminated!). Generally, parallel speed-up will be initially somewhat linear with number of processors used, but will eventually decay (see figure 3.8).

For example, the mesh used for the channel flow DNS examined in chapter 7 has `N_Z` $= 80$. Then the maximum number of processes which could be employed for parallel execution is 40. However, the number of elements (96) is rather small, so one could perhaps more efficiently use 20, or 10 (and maybe even as few as 8, 4 or 2) processors: one has to check the speed-up to see which is most efficient, consider how long you are prepared to wait for results and how many processors are available for use.

Here is how to use `dns_mp` for that problem with 8 processors:

```
$ mpirun -np 8 dns_mp chan > /dev/null
```

(Redirection of `stdout` to `/dev/null` is not necessary, but once you are sure that the problem will run OK, is generally a Good Idea.)

Apologies, but for now at least, none of the *Semtex* utilities will run in parallel.

Chapter 4

Extra controls

This chapter describes some additional features that are implemented within the Navier–Stokes solver `dns` to control execution and output.

4.1 Default values of flags and internal variables

There are two simple ways to establish the default values of all the internal flags and variables used by *Semtex*. The first is via the `calc` utility: run `calc -h` and check the output (this will also show you all the functions available to the parser for calculating `TOKEN` variables, initial and boundary conditions). The second is to examine the file `femlib/defaults.h`.

4.2 Checkpointing

By default, intermediate solutions are written out as checkpoint dumps in file `session.chk` every `IO_FLD` steps (default value `IO_FLD = 500`), rotating this to `session.chk.bak` so there are usually two checkpoint files available for restarting if execution stops prematurely (e.g. if terminated by a queuing system or by a floating point error). Once the final time (`N_STEP`) is reached, the outcome (the terminal solution field) is written to `session.fld`.

Sometimes however, one wants a sequence of field dumps to be written to `session.fld`. One can toggle this behaviour on the command line using `dns -chk`, or alternatively set the `TOKEN` `CHKPOINT=0` (the default being `CHKPOINT=1`). Beware: turning off checkpointing can result in the generation of extremely large `session.fld` files.

4.3 Iterative solution

Two matrix solution methods are implemented for Helmholtz problems associated with the viscous substep of the time splitting. By default, direct Schur-complement solutions are used. The associated global matrices can consume quite large amounts of memory, typically much more than is required for storage of the associated field variable. Iterative (PCG) solution can also be selected, and this has the advantage that since it is matrix-free, no global matrices are required, however, solution may be slower than for the direct solver (depending on the condition number of the global matrix problem).

The token that controls the selection of matrix solution method is `ITERATIVE`. For `dns`, PCG solution can be selected for the viscous substep of the solution (`ITERATIVE = 1`). This can also be selected via a command-line option (`dns -i`), but note that this overridden by tokens set in the `session` file (the default value is `ITERATIVE = 0`).

Iterative solution can be useful for the viscous substep, particularly when the Reynolds number is high, since this decreases the condition number of the associated global matrices. In fact, iterative solutions for the viscous substep can execute faster than direct solutions at high Reynolds number,

although this is platform dependent. You should always consider trying `ITERATIVE = 1` as an option for simulations where the Reynolds number is more than a few hundred.

4.4 Alternative forms of nonlinear terms

Various forms of the nonlinear terms for the Navier–Stokes equations (e.g. $\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}$) are available. The more standard ones are generally selectable from the `dns` command line, but additional forms can be selected via the token `ADVECTION` (which has default value 1). If you wish to solve the (unsteady) incompressible Stokes equations for which the nonlinear terms are zero, set `ADVECTION = 5` (or use `dns -N`). Owing to various vector identities and incompressibility, $\boldsymbol{\nabla}\cdot\boldsymbol{u}=0$, the various nonlinear terms are equivalent in the continuum setting:

$$\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}\equiv\boldsymbol{\nabla}\cdot\boldsymbol{u}\boldsymbol{u}\equiv\tfrac{1}{2}[\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}+\boldsymbol{\nabla}\cdot\boldsymbol{u}\boldsymbol{u}]\equiv\boldsymbol{u}\times\boldsymbol{\nabla}\times\boldsymbol{u}\equiv\boldsymbol{u}\times\boldsymbol{\nabla}\times\boldsymbol{u}-\tfrac{1}{2}\boldsymbol{\nabla}\boldsymbol{u}\cdot\boldsymbol{u}.$$

The five forms above are generally named: convective (or, sometimes, non-conservative), conservative, skew-symmetric, rotational-1 and rotational-2. While equivalent in the continuum setting, they have somewhat distinct behaviours in the discrete (numerical) setting.

The convective form (standard in fluid mechanics texts) is generally the simplest, and cheapest to compute (use `ADVECTION = 2` or `dns -C`). Going back (at least) to the work of Zang (1991), it is generally acknowledged that the skew-symmetric form has superior energy conservation properties in the discrete setting and in our experience is generally the most benign when solution resolution starts to become marginal. In early versions of *Semtex*, this was the default form (use `ADVECTION = 0` or `dns -S`). It does, however, cost double to compute compared to the convective form. A simple variation, suggested by Kerr (1985), is to use the convective and conservative forms on alternate timesteps — this is called the 'alternating' skew-symmetric form; 'on average' one may expect similar behaviour to the full skew-symmetric form, and it costs much the same to compute as the convective form. In our experience it is about as robust as full skew-symmetric form, and since it is cheaper, it has become the default option in *Semtex* (use `ADVECTION = 1` to explicitly select it). It does, however, have a slight defect: owing to alternation, it can produce small high-frequency temporal oscillations in the solution, of period $2\Delta t$. It may also slightly reduce the asymptotic convergence rates of steady solutions. If you are seeking very clean solutions at high resolution (e.g. to compute base flows for stability analysis, or your flow may have sensitive dynamics), it is best to use either convective or full skew-symmetric forms. In general, or perhaps when computing, say, a turbulent flow, which is full of disturbance, you might profitably use the default.

The two rotational forms are provided for completeness if you have a specific interest in them — use either `ADVECTION = 3` or `4`. Their computational cost is similar to the conservative form. In summary:

`ADVECTION = 0`	$\frac{1}{2}[\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}+\boldsymbol{\nabla}\cdot\boldsymbol{u}\boldsymbol{u}]$	`dns -S`
`ADVECTION = 1`	$[\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}]^{(n)}$ or $[\boldsymbol{\nabla}\cdot\boldsymbol{u}\boldsymbol{u}]^{(n+1)}$	(default)
`ADVECTION = 2`	$\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}$	`dns -C`
`ADVECTION = 3`	$\boldsymbol{u}\times\boldsymbol{\nabla}\times\boldsymbol{u}$	
`ADVECTION = 4`	$\boldsymbol{u}\times\boldsymbol{\nabla}\times\boldsymbol{u}-\frac{1}{2}\boldsymbol{\nabla}\boldsymbol{u}\cdot\boldsymbol{u}$	
`ADVECTION = 5`	0	`dns -N`

You may ask: what about the conservative form, $\boldsymbol{\nabla}\cdot\boldsymbol{u}\boldsymbol{u}$? Isn't that as cheap to compute as the convective form? Well, yes it is, but Wilhelm and Kleiser (2001) showed that it was also linearly unstable in a discrete setting. And, during development, our companion stability analysis code, `dog`, produced erroneous results when the skew-symmetric form of advection terms was initially employed. So, the conservative form is not provided as an option for `dns` and indeed, for this reason, the linearised advective term used by `dog` ($\boldsymbol{u}'\cdot\boldsymbol{\nabla}\boldsymbol{U}+\boldsymbol{U}\cdot\boldsymbol{\nabla}\boldsymbol{u}'$) is based on the convective form.

Prior to *Semtex* V8, nonlinear product terms were dealiased in the Fourier direction (only) when using the serial version of `dns`. Partly for consistency across serial and parallel computations, and

partly for simplicity, dealiasing was subsequently removed from the code. This can somewhat degrade the rate of exponential convergence of results in the Fourier direction for serial execution, compared to earlier releases. Results for parallel execution are unchanged from before, since no dealiasing was ever employed for parallel executables. We note that in lieu of dealiasing, a weak application of spectral vanishing viscosity (see § 4.12) can have beneficial effects at high wavenumbers when resolution is marginal (typically, when Reynolds numbers are high).

4.5 Wall fluxes, forces, torques

A file called `session.flx` is used to store the integral over the `wall` group boundaries of viscous and pressure stresses (i.e. lift and drag forces). Output is done every `IO_HIS` steps. For each direction (x, y, z), the outputs are in turn the pressure, viscous, and total force per unit length (in z). In 2D the z-components are always zero, while in 3D the z-component pressure force is always zero, owing to the fact that the geometry is invariant in that direction. For cylindrical geometries, the output values are forces per radian (in the x and y directions) and torque per radian (in the z direction) rather than forces per unit length.

If a scalar (`c`) is present in the simulation, then its integral flux is also computed over the `wall` group boundaries and output in the `session.flx` file, preceding the wall tractions.

4.6 Wall tractions

If the token `IO_WSS` is set to a non-zero value then the normal and the single (2D) or two (3D) components of tangential boundary traction are computed on the `wall` group, and output every `IO_WSS` steps in the file `session.wss`. This is a binary file with structure similar to a field dump. The utility `wallmesh` is used to extract the corresponding mesh points along the walls (and can be used with `sem2tec` to produce *Tecplot* input files). Note also that there is a stand-alone utility called `traction` which is a post-processor that takes a standard `.fld` file and produces a wall traction file.

4.7 Modal energies

For three-dimensional simulations (`N_Z > 1`), a file of modal energies, `session.mdl`, is produced. This provides valuable diagnostic information for turbulent flow simulations. For each active Fourier mode k in the simulation, the value output every `IO_HIS` steps is

$$E_k = \frac{1}{2A} \int_\Omega \hat{\boldsymbol{u}}_k^* \cdot \hat{\boldsymbol{u}}_k \, \mathrm{d}\Omega,$$

where A is the area of the 2D domain Ω. (In cylindrical coordinate problems, the integrand is multiplied by radius.) Each line of the file contains the time t, mode number k and E_k.

Note that the *energies are output only for non-negative Fourier modes*. To get the correct estimates for the one-sided spectrum (and to satisfy Parseval's relation), the energies for non-zero modes should be doubled.

4.8 History points

History points are used to record solution variables at fixed spatial locations as the simulation proceeds. The locations need not correspond to grid points, as data are interpolated onto the given spatial locations using the elemental basis functions. Locations of history points are declared in the `session` file as follows:

```
<HISTORY NUMBER=1>
#       tag     x       y       z
        1       0       0       0
</HISTORY>
```

A file called `session.his` is produced as output. Each line of the file contains the step number, the time, the history point tag number, followed by values for each of the solution variables. The step interval at which history point information is dumped to file is controlled by the `IO_HIS` token; the default value is `IO_HIS = 10`.

Regarding interpolation: to locate a desired history point in the domain, the solver uses a Newton–Raphson iteration. It is not impossible that that this can fail to converge; if it does so, you can try adjusting `TOKEN`s `NR_MAX` and/or `TOL_POS` from their default values (20 and 1×10^{-5}). See also the discussion under § 6.9. The same considerations in fact apply to some other related functions that involve data extraction at points, such as the `probe` utility and particle tracking.

4.9 Averaging

Set `AVERAGE = 1` in the tokens section to get averages of field variables left in files `session.ave` and `session.avg` (which are analogous to `session.chk` and `session.fld`, but `session.ave.bak` is not produced). Averages are updated every `IO_HIS` steps, and dumped every `IO_FLD` steps. Restarts are made by reading `session.avg` if it exists.

Setting `AVERAGE = 2` will accumulate averages for Reynolds stresses as well, with reserved names `ABCDEF`, corresponding to products

```
uu uv uw     A  B  D
   vv vw  =     C  E
      ww           F
```

The hierarchy is named this way to allow accumulation of products in 2D as well as 3D (for 2D you get only `ABC`). In order to actually compute the Reynolds stresses from the accumulated products you need to run the `rstress` utility, which subtracts the products of the means from the means of the products:

```
rstress session.avg > reynolds-stress.fld
```

An alternative function of `rstress` is to subtract one field file from another:

```
rstress good.fld test.fld | convert | diff
```

With the addition of scalar as well as velocity components, the averaging (and naming) is extended in the following way:

```
uu uv uw  uc    A  B  D  G
   vv vw  uv =     C  E  H
      ww  uw          F  I
          cc             J,
```

again with a fairly obvious two-dimensional restriction (`ABCGHJ`). Again, use `rstress` to subtract products of means from means of products.

Setting `AVERAGE = 3` will accumulate sums of additional products for computation of terms in the kinetic energy transport equation. You will then need to use the `eneq` utility to actually compute the terms. Presently this part of the code is only written for Cartesian coordinates, and only does collections based on kinetic energy.

4.10 Phase averaging

Phase averaging is useful for turbulent flows with a dominant (and known) underlying temporal period. We can collect statistics (with `AVERAGE=1`, 2 or 3) — much as for the case without phase averaging enabled — conditional on phase in the cycle of the underlying period, see Reynolds and Hussain (1972). Turning on phase averaging does not preclude or stop collection of standard statistics. The enabling token is `N_PHASE`, which must be a positive integer; in addition one needs token `STEPS_P` (steps per period) which must be chosen such that `STEPS_P` modulo `N_PHASE` is zero, and also `N_STEP` modulo `N_PHASE` must be zero *and* `IO_FLD=STEPS_P/N_PHASE`. Statistics are written to files `session.0.phs` ... `session.X.phs` where `X=N_PHASE-1`. The Reynolds stresses computed from these files will represent fluctuations around the conditional average flow at each phase point (the so-called 'triple decomposition').

The slight difficulty is that if the period is not very well-defined or we have a poor estimate of it, our sampling phase will slowly drift unless we take corrective action. However if the underlying period is very well defined (e.g. the flow is periodically forced) the method has great potential.

4.11 Particle tracking

The code allows for tracking of massless particles, but this only works correctly for non-concurrent execution at present. Tracking is quite an expensive operation, since Newton–Raphson iteration is used to relocate particles within each element at every timestep.

The application looks for a file called `session.par`. Each line of this file is of form

```
#     tag  time  ctime  x     y      z
      1    0.0   0.0    1.0   10.0   0.5.
```

The `time` value is the integration time, while `ctime` records the time at which integration was initialised.

Output is of the same form, and is called `session.trk`. The use of separate files, rather than by declaration in the session file, is intended so that `session.trk` files can be moved to `session.par` files for restarting. Particles that aren't in the domain at startup, or leave the domain during execution, are deleted.

Setting `SPAWN = 1`, re-initiates extra particles at the original positions every timestep. With spawning, particle tracking can quickly grow to become the most time-consuming part of execution.

4.12 Spectral vanishing viscosity (SVV)

Spectral vanishing viscosity amounts to implementing larger viscosity at higher wavenumbers either in Fourier space or in spectral element polynomial space. The idea is that as resolution is increased via p-refinement, the effect 'vanishes' (Tadmor; 1989; Maday et al.; 1993). One may regard SVV either as a type of implicit large-eddy simulation methodology (Pasquetti; 2006) or as a means of stabilising spectral element solutions especially at high Reynolds numbers (Xu and Pasquetti; 2004; Kirby and Sherwin; 2006), effectively as a palliative used in lieu of dealiasing. Neither of these ideas has firm theoretical underpinning at this stage, yet the method does appear quite effective in reducing resolution requirements for turbulent flow simulations (Koal et al.; 2012; Chin et al.; 2015). Our implementation and nomenclature follows the 'standard method' described in Koal et al. (2012). One can turn on SVV separately and with different parameters for (x, y) spectral elements and in the Fourier (z) direction. These are all declared in the `TOKENS` section.

`SVV_MN` Corresponds to cut-in mode M_{zr} in spectral elements. Must be less than `N_P`.
`SVV_MZ` Corresponds to cut-in mode M_{φ} in Fourier direction. Must be less than `N_Z/2`.

SVV_EPSN Corresponds to ε_{zr}. Should be a value larger than KINVIS, e.g. 5*KINVIS.
SVV_EPSZ Corresponds to ε_{φ}. A value larger than KINVIS.

The default polynomial transform in to place spectral element expansions into a discrete hierarchical space is the discrete Legendre transform (see e.g. Blackburn and Schmidt; 2003). This can be changed in `src/svv.cpp`.

4.13 General body forcing

This extension was largely developed by Thomas Albrecht.

If found, the FORCE section of the session file allows you to declare various types of body forcing, i.e. add a source term to the RHS of the Navier–Stokes equation. The currently implemented types include (any combination allowed):

$\boldsymbol{f} =$	$\boldsymbol{f}_{const}$	constant force (frame acceleration)
$+$	$\boldsymbol{a_1}(\boldsymbol{x})$	steady, but spatially varying force
$+$	$\boldsymbol{a_2}(\boldsymbol{x})\,\boldsymbol{\alpha}(t)$	modulated force
$-$	$m_1(\boldsymbol{x})\,(\boldsymbol{u}-\boldsymbol{u}_0)$	sponge region
$-$	$m_2(\boldsymbol{x})\,(\boldsymbol{u}/\lvert\boldsymbol{u}\rvert)\,\lvert\boldsymbol{u}(\boldsymbol{x},t)\rvert^2$	'drag' force
$+$	$\boldsymbol{\epsilon}G$	white noise
$-$	$\chi(\boldsymbol{u}-\overline{\boldsymbol{u}})$	selective frequency damping
$-$	$2\,\boldsymbol{\Omega}\times\boldsymbol{u}-(\mathrm{d}\boldsymbol{\Omega}/\mathrm{d}t)\times\boldsymbol{x}-\boldsymbol{\Omega}\times\boldsymbol{\Omega}\times\boldsymbol{x}$	Coriolis force
$-$	$\beta_T(c-c_{\mathrm{ref}})\boldsymbol{g}$	Boussinesq buoyancy.

4.13.1 Constant force

For example, a force constant in time and space $\boldsymbol{f} = \boldsymbol{f}_{const}$ is declared by:

```
<FORCE>
        CONST_X = 4
        CONST_Y = 0
        CONST_Z = 0
</FORCE>
```

This type of forcing must not be time or space dependent. It is suitable for periodic channel flow, where you have a uniform and steady force driving the flow, see `channel-FX` for an example session.

All forcing terms are applied in physical space.

Unless otherwise noted, any skipped keyword defaults to 0. Any line starting with a hash # is ignored.

4.13.2 Steady force

A spatially varying, steady force $\boldsymbol{f} = \boldsymbol{a}(\boldsymbol{x})$, computed (or read from a file) during pre-processing and applied every time step. See `box-steady` for the complete example session. It suits applications requiring localised, steady forcing.

```
<FORCE>
        STEADY_X =  cos(x)
        STEADY_Y = -sin(z)
        STEADY_Z = -cos(y)
        # STEADY_FILE = box-steady.force.fld
</FORCE>
```

You may also point STEADY_FILE to a field file, in which case the force is taken from the uvw fields of that file and STEADY_[XZY] is ignored.

4.13.3 Modulated force

A spatially varying force, which is modulated in time, $\boldsymbol{f} = \boldsymbol{a}(\boldsymbol{x})\,\boldsymbol{\alpha}(\boldsymbol{t})$. The steady part $\boldsymbol{a}(\boldsymbol{x})$ is computed (or read from a file) during pre-processing, while $\boldsymbol{\alpha}(\boldsymbol{t})$ is evaluated each time step.

```
<FORCE>
        # -- spatially varying part
        MOD_A_X =  cos(x)
        MOD_A_Y = -sin(z)
        MOD_A_Z = -cos(y)
        # MOD_A_FILE = box-mod.force.fld

        # -- time varying part
        MOD_ALPHA_X = step(t, 10)
        MOD_ALPHA_Y = step(t, 10)
        MOD_ALPHA_Z = step(t, 10)
</FORCE>
```

4.13.4 Sponge region

This implements a so-called 'sponge region' defined by the shape function $m(x)$ in which a (physically meaningless) penalty term $\boldsymbol{f} = m(\boldsymbol{x})\,(\boldsymbol{u} - \boldsymbol{u}_0)$ forces the flow towards a given solution $\boldsymbol{u}_0$. It is especially useful for inflow–outflow simulations of vortex shedding or turbulence: if the velocity fluctuations hit the outflow boundary condition, they cause unphysical reflections back into the domain which distort the upstream flow. A sponge region placed just upstream the outflow boundary helps to reduce the velocity fluctuations to (near) zero and thereby prevents those reflections. The following section would apply the penalty term for $20 \leq x \leq 24$, and within that region forces the velocity to approach $(1, 0, 0)$. That given solution may be a function of space, but must be steady.

```
<FORCE>
        SPONGE_M = 5. * step(x,20)*heav(24-x)
        SPONGE_U = 1
        SPONGE_V = 0
        SPONGE_W = 0
</FORCE>
```

4.13.5 'Drag' force

An approximate drag force $\boldsymbol{f} = -m(\boldsymbol{x})\,(\boldsymbol{u}/|\boldsymbol{u}|)\,|\boldsymbol{u}(\boldsymbol{x}, t)|^2$. Be aware that we use the previous time step's velocity $\boldsymbol{u}^n$ here.

```
<FORCE>
        DRAG_M = heav((x-2)^2 + y^2)
</FORCE>
```

4.13.6 White noise force

Similar to the noiz tool, this continuously adds random perturbation $\boldsymbol{f} = (\epsilon_x, \epsilon_y, \epsilon_z)^T\,G$ in specified direction, where G is a normally distributed random variable. It is now only possible to apply the forcing uniformly in physical space.

The following example applies white noise in $x-$direction to mode 0:

```
<FORCE>
        WHITE_EPS_X = 0.1
        WHITE_EPS_Y = 0
        WHITE_EPS_Z = 0
</FORCE>
```

Adding white noise in all three directions degrades performance by about 10%.

4.13.7 Selective frequency damping (SFD)

This is a means of obtaining an approximate steady state solution to the Navier–Stokes equations using an unsteady solver, originally described by Åkervik et al. (2006). SFD applies a penalty term of the form $-\chi(\boldsymbol{u}-\overline{\boldsymbol{u}})$ to the right-hand side of the momentum equations, where $\overline{\boldsymbol{u}}$ is an estimate of the time-mean solution that is updated as integration proceeds (and held in internal storage). SFD can also be considered as applying an IIR low-pass digital filter to the discrete approximation of the Navier–Stokes equations.

The two parameters are `SFD_CHI` (i.e. penalisation multiplier χ) and `SFD_DELTA`, which is the time constant Δ used in updating a forwards-Euler approximation of the steady flow $\overline{\boldsymbol{u}}$ (see reference). Both values are problem-specific and should be tuned to get acceptable results. Note that it is not always possible to obtain a steady outcome with SFD, and that it is generally preferable to use standard skew-symmetric form of the nonlinear terms for `dns`, rather than the now-default alternating skew symmetric form: this can be achieved by setting token `ADVECTION = 0` or by using command-line flag `-S` with `dns`.

```
<FORCE>
        SFD_CHI   = 0.2
        SFD_DELTA = 0.75
</FORCE>
```

A final point to note is that the values of the two parameters are parsed once, at the beginning of runtime, after any restart file (if present) is read. Thus, their definitions can contain the temporal variable `t`, but the value used is whatever holds at the start of runtime.

4.13.8 Rotating frame of reference: Coriolis and centrifugal force

If the flow is to be computed in a rotating frame of reference, additional acceleration terms appear, namely $\boldsymbol{f} = -2\,\boldsymbol{\Omega}\times\boldsymbol{u} - (\mathrm{d}\boldsymbol{\Omega}/\mathrm{d}t)\times\boldsymbol{x} - \boldsymbol{\Omega}\times(\boldsymbol{\Omega}\times\boldsymbol{x})$ (Batchelor; 1967). The vector of rotation $\boldsymbol{\Omega}$ is *always* given in Cartesian co-ordinates, even if `CYLINDRICAL = 1`. Its magnitude and/or orientation can change with time. However, the axis of rotation it is always assumed to go through the origin. Depending on whether $\boldsymbol{\Omega}$ is steady or not, usage slightly differs.

For unsteady $\boldsymbol{\Omega}$, set the flag `CORIOLIS_UNSTEADY = 1` and give $\boldsymbol{\Omega}$ and $\mathrm{d}\boldsymbol{\Omega}/\mathrm{d}t$. All terms are re-evaluated each time step.

```
<TOKENS>
        f        = 1.
        omega    = TWOPI * f
</TOKENS>

<FORCE>
        CORIOLIS_UNSTEADY = 1

        CORIOLIS_OMEGA_X = 0
        CORIOLIS_OMEGA_Y = 0
```

```
        CORIOLIS_OMEGA_Z = omega * sin(t)

        CORIOLIS_DOMEGA_X_DT = 0
        CORIOLIS_DOMEGA_Y_DT = 0
        CORIOLIS_DOMEGA_Z_DT = omega * cos(t)
</FORCE>
```

For constant $\boldsymbol{\Omega} \neq f(t)$, the term $-(\mathrm{d}\boldsymbol{\Omega}/\mathrm{d}t) \times \boldsymbol{x}$ vanishes, and the centrifugal force $-\boldsymbol{\Omega} \times (\boldsymbol{\Omega} \times \boldsymbol{x})$ can be computed during pre-processing. Set `CORIOLIS_UNSTEADY = 0` and make sure to include the centrifugal force manually using a steady force as it is no longer computed automatically[1]. A temporal derivative $\mathrm{d}\boldsymbol{\Omega}/\mathrm{d}t$, if given, is ignored.

```
<FORCE>
        CORIOLIS_UNSTEADY = 0

        CORIOLIS_OMEGA_X = 0
        CORIOLIS_OMEGA_Y = 0
        CORIOLIS_OMEGA_Z = omega

        # -- centrifugal term for Omega = (0, 0, omega)^T
        #    for a cylindrical problem
        STEADY_X = x*omega^2
        STEADY_Y = omega^2*y*(cos(z)^2)
        STEADY_Z = -omega^2*y*cos(z)*sin(z)
</FORCE>
```

See Albrecht et al. (2015) for an example of DNS carried out in a rotating frame of reference.

4.13.9 Boussinesq buoyancy

See the example of § 3.5. This option is available if your simulation includes a scalar, in which case it can be considered as a temperature, allowing for the introduction of body forces based on density variation produced by a coefficient of thermal expansion β_T. In the Boussinesq approximation, the flow is still assumed to be incompressible, and the added body force is of form $-\beta_T(c - c_{\mathrm{ref}})\boldsymbol{g}$, where c_{ref} is a reference value of scalar (i.e. temperature) and $\boldsymbol{g}$ is the gravitational acceleration vector. Note that for an ideal gas, $\beta_T = 1/c_{\mathrm{ref}}$.

```
<FORCE>
        BOUSSINESQ_TREF    = 288.15
        BOUSSINESQ_BETAT   = 1./BOUSSINESQ_TREF
        BOUSSINESQ_GRAVITY = 10.
        BOUSSINESQ_GY      = -1.0
</FORCE>
```

The gravity vector is supplied as a magnitude, `BOUSSINESQ_GRAVITY`, and a set of direction cosines; in the above example, only the y component is set, with the other components defaulting to zero. If cylindrical coordinates are employed, only the axial, x, direction cosine is used, i.e. the gravity vector must be aligned with the axis of the coordinate system. For historical reasons, the scalar c goes under the pseudonym `T` here, i.e. $c_{\mathrm{ref}} \equiv$ `BOUSSINESQ_TREF`

[1] If you're lazy, or for cross-checking, you could set `CORIOLIS_UNSTEADY = 1` and omit `CORIOLIS_DOMEGA_[XYZ]_DT` to have the centrifugal term computed automatically. Note, however, that this degrades performance as it is done each time step.

A further restriction to note is that while, logically, buoyancy forces should recognise potential-type reference frame forces (such as rotation), at present this is not implemented, so that buoyancy only responds to gravity fields.

As of 2022, this kind of gradient-based Boussinesq treatment has been deprecated in favour of the more complete approach described in the following section.

4.13.10 'Canonical' steady Boussinesq buoyancy

This is the treatment of Boussinesq buoyancy described in Blackburn et al. (2021), whereby density fluctuation is applied to all the convective and frame-acceleration terms in the Navier–Stokes equations. In fact, here, it is applied to the convective, frame-acceleration, and any other forcing terms. (It is usual in GFD Boussinesq applications to apply density variation only to gravitational body force — equivalent to a frame acceleration — on the assumption that this is typically large compared to the fluid acceleration terms.)

Following from Blackburn et al. (2021), the canonical Boussinesq treatment of the incompressible Navier–Stokes equations (§ 2.2) is (with body force terms $\boldsymbol{f}$ included, and transposed onto the LHS of the momentum equation)

$$\frac{\rho}{\rho_0}\left[\frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t}-\boldsymbol{f}\right]=\left(1+\frac{\rho'}{\rho_0}\right)\left[\frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t}-\boldsymbol{f}\right]=-\frac{1}{\rho_0}\boldsymbol{\nabla}p+\nu\nabla^2\boldsymbol{u},\quad \boldsymbol{\nabla}\cdot\boldsymbol{u}=0, \tag{4.1}$$

where $\boldsymbol{u}$ is the velocity field measured in an inertial frame of reference. Using the typical approach where the effect of density variation is dropped from the local accelerative term, the momentum equation above is approximated as

$$\frac{\partial\boldsymbol{u}}{\partial t}+\left(1+\frac{\rho'}{\rho_0}\right)\left[\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}+2\boldsymbol{\Omega}\times\boldsymbol{u}+\boldsymbol{\Omega}\times(\boldsymbol{\Omega}\times\boldsymbol{r})+\boldsymbol{\alpha}\times\boldsymbol{r}+\boldsymbol{A}-\boldsymbol{f}\right]=-\frac{1}{\rho_0}\boldsymbol{\nabla}p+\nu\nabla^2\boldsymbol{u}, \tag{4.2}$$

where now $\boldsymbol{u}$ is measured relative to the origin of the arbitrarily accelerating and rotating frame of reference. As in our standard treatment of Boussinesq buoyancy (§ 4.13.9) the relative density variation $\rho'/\rho_0=\beta_T(T_{\mathrm{ref}}-T)\equiv\beta_T(c_{\mathrm{ref}}-c)$. Equation (4.2) is the 'canonical steady Boussinesq' treatment of buoyancy terms. As explained in Blackburn et al., it is a robust treatment of Boussinesq buoyancy and fulfils the original intention in the case of steady flows — and the omission of effect of density variation on the local acceleration term is also a part of standard Boussinesq treatments, so in effect it is a more complete approach than the standard method. Note that the flows considered need not be steady; it is just the case that, as for standard Boussinesq treatments, the approximation is more complete when they are. We call the method the convective–frame-acceleration Boussinesq treatment (that's a mouthful, so we use the acronym CFB below).

As outlined in Blackburn et al., a gravitational field $\boldsymbol{g}$ would be included as a frame acceleration term $\boldsymbol{A}\equiv-\boldsymbol{g}$ and dealt with using the approach described in § 4.13.1. Any frame-rotation terms would be dealt with as described in § 4.13.8.

To use CFB buoyancy, one needs to define `CFB_BETA_T` and `CFB_T_REF` in the `FORCE` section. These are equivalent to the regular Boussinesq tokens outlined in the previous section. In addition, one may need to apply frame acceleration terms as outlined above in §§ 4.13.1, 4.13.2 and 4.13.8.

For example, consider the case dealt with in Blackburn et al. (2021) § 3.1, see the present figure 4.1(*a*). This is an axisymmetric rotating flow dealt with in cylindrical coordinates. It has frame acceleration in the $-x$ direction, which is equivalent to a steady body force in the $+x$ direction. As explained in § 4.13.8, we can set `CORIOLIS_UNSTEADY=0` and explicitly add a steady (centrifugal) body force $|\boldsymbol{\Omega}|^2 r$ — this is computationally cheaper than computing $-\boldsymbol{\Omega}\times(\boldsymbol{\Omega}\times\boldsymbol{x})$ at every timestep. We still need to set `CORIOLIS_OMEGA_X` to ensure the Coriolis body force terms $-2\boldsymbol{\Omega}\times\boldsymbol{u}$ are computed. Here is the `FORCE` section for that example (see also session file `MMBL` in the `mesh` directory). The supplied values in various of the assignments below correspond to user-defined `TOKENS`.

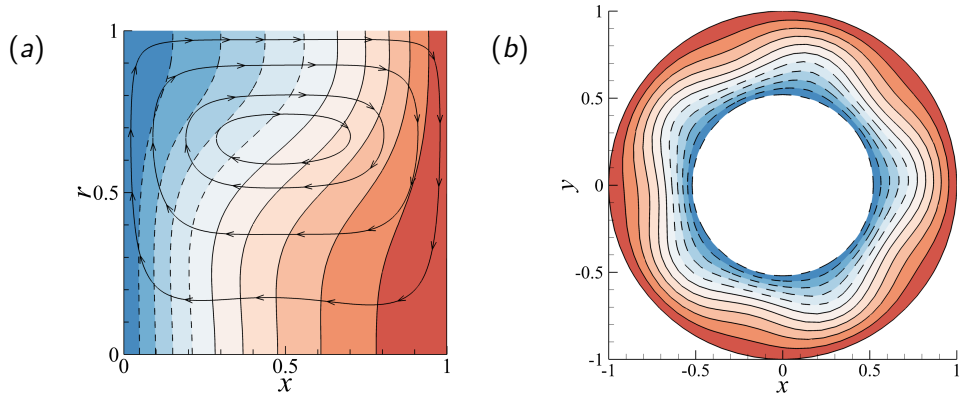


Figure 4.1: Canonical steady buoyancy test case examples from Blackburn et al. (2021). Red/blue colours indicate warm/cool fluid. See supplied session files `MMBL` and `annulus36`.

```
<FORCE>
    CFB_BETA_T         = BETA_T
    CFB_T_REF          = T_REF
    CORIOLIS_UNSTEADY  = 0
    CORIOLIS_OMEGA_X   = OMEGA
    STEADY_Y           = OMEGA*OMEGA*y
    CONST_X            = GRAVITY
</FORCE>
```

Another example is the test case of Blackburn et al. (2021) § 3.2, see present figure 4.1(*b*). This is a rotating flow, and the geometry and boundary conditions are again axisymmetric, but the problem is dealt with in Cartesian coordinates (which allows the $k = 5$ wavy primary instability to develop in 2D). There is no gravitational acceleration in this case. Note the treatment of the centrifugal force terms. See also the session file `annulus36` in the `mesh` directory.

```
<FORCE>
    CFB_T_REF          = T_REF
    CFB_BETA_T         = BETA_T
    CORIOLIS_UNSTEADY  = 0
    CORIOLIS_OMEGA_Z   = OMEGA
    STEADY_X           = OMEGA*OMEGA*x
    STEADY_Y           = OMEGA*OMEGA*y
</FORCE>
```

By default, hydrostatic pressure gradients associated with steady body forces or accelerations are removed for CFB. This is convenient in the case that there are free boundaries and one does not wish to apply associated pressures. If you wish hydrostatic terms to appear, you should also set `CFB_REMOVE_HYDRO=1`.

Chapter 5

Code design and the Semtex API

While the top level of the code is written in C++, the bulk of computational work is carried out using 3rd-party libraries: BLAS and LAPACK (vendor- or distribution-supplied) for two-dimensional operations, Temperton's 2-3-5 prime factor FFT (incorporated into `femlib`) for three-dimensional operations, and (open)MPI for parallel operations. Depending on what your application hits the hardest, one or other of these things may be the speed-determining component. Optimising compilers are liable to make a only a small difference — most speed-up is now to be achieved by further algorithm development. However, it is definitely worthwhile seeking out a fast BLAS library (as supplied with Apple's Xcode, Intel's MKL, or AMD's ACML).

Some fundamental design decisions:

1. In *Semtex*, all floating point numbers are double precision variables.
2. Most real-type data arrays are flat (one-dimensional). In C and C++, such arrays have indices starting at 0, while in Fortran, indexing starts at 1 by default — though the start address is identical. This makes them easy to use with Fortran routines, but with the following caveat.
3. The layout of these flat arrays are typically taken as row-major, which is standard C/C++. However, Fortran uses column-major ordering. For this reason, any BLAS or LAPACK matrix operation may at first sight seem to be the transpose of what is intended.
4. Ordering of storage in data `AuxFields` is, working bottom-up: (x, y) data in each element with internal row-major order starting from the bottom-left corner (i.e. at the first vertex), then a list of elements as declared in the `ELEMENTS` section of the session file, then by $z-$planes.
5. Most low-level C++ methods in the code do not incorporate internal data storage but can be regarded as operator routines, and get handed addresses to appropriate parts of storage from within the flat data arrays on which to work.

Coding conventions:

1. Class private or protected member data names start with an underscore, e.g. `_ntot`.

5.1 Useful things to know about

Hierarchy of main data storage From top down: `Domain`, `Fields`, `AuxFields`, `Elements`. The key entity for most applications programming is the `AuxField` class, which contains scalar field variables and a list of `Element` pointers. The `Field` class inherits from the `AuxField` class, adding a list of boundary condition applicators and the ability to solve elliptic problems. The `Domain` class holds an array of `Field` pointers and most of its internal storage is publicly accessible.

The Geometry class This class (`src/geometry.cpp`) centralises information about the 'logical' (rather than 'spatial') geometry of the solution domain, for example, the number of points along the edge of each element, the number of elements, number of processes, the total number of z-planes, the number of z-planes per process, the number of Fourier modes, the number of data in a single (x, y) plane, etc.

Most of these are fairly straightforward (peruse `geometry.h`), but it may be worthwhile to explain the distinction between `Geometry::nPlane()` and `Geometry::planeSize()`. The first of these is simple to understand: the number of elements multiplied by the number of points in each element. The second is similar, but often somewhat larger, because of (a) restrictions brought on by the FFT used (the number of points in a plane must be even if the solution is three-dimensional and an FFT will be used in the orthogonal direction) and (b) further requirements for parallel execution, which requires block-transposes of data across processes, as explained in Rudman and Blackburn (2006). In each case, the extra padding does not end up entering computations. The amount of data allocated internally for each data plane is `Geometry::planeSize()`. A consequence of this distinction is that the memory for the first storage location in any data plane does not necessarily immediately follow that for the last storage location of the preceding plane. In an `AuxField`, the variable `_plane` references a set of pointers to the first storage location of each plane on the current process.

Timestepping algorithm, storage schemes The time integration scheme used in `dns` is the 'stiffly stable' algorithm, based on backward differencing in time, and uses time-splitting as originally described in Karniadakis et al. (1991). What is stored where, and when — see figure 5.1.

Support libraries and static member functions The two base-class libraries provided as part of *Semtex* are called `veclib` and `femlib`. `Veclib` provides vector algebra primitives and can be regarded as an extension to the BLAS: check `veclib/README` for a summary of the operations it provides. `Femlib` is a bundle of low-level operations that are finite-element and spectral-element-shape function related, Fourier transforms, and a function string parser (see `femlib/initial.y`).

It is standard to use library-name prefixes on calls to these libraries, as well as for LAPACK and BLAS calls (`Veclib::`, `Femlib::`, `Lapack::`, `Blas::`) to help the programmer identify the libraries involved.

BLAS-conformant increments Where access to a block of data is obtained on the offset, skip model, this is done in a BLAS-conformant way if the skip is negative: the start address is at the high end of the memory space.

Implications of Fourier transform in one direction For three-dimensional computations, note that for all of the timestepping loop, other than during computation of the nonlinear terms, field variables are held in the Fourier-transformed state. Field data written out to file are in physical space.

Static condensation Because finite/spectral element shape functions can be partitioned into sets which do/do not have non-zero support at the edges of elements, and because global Galerkin shape functions have only comparatively local support (confined to mating finite elements), only element-edge variables need to be considered when assembling the global Galerkin forms of elliptic equations. Subsequently, remaining element-internal variables can be obtained by local solutions on an element-by-element basis. Recognition of this fact has long been a part of finite-element procedures, where it goes under various names: substructuring; static condensation; Schur-complement solution. *Semtex* uses this methodology for assembly and solution of elliptic equations when direct methods are used.

	Domain				Us			Uf		
STAGE	D->u[0] u	D->u[1] v	D->u[2] w	D->u[3] p	Us[.][0]	Us[.][1]	Us[.][2]	Uf[.][0]	Uf[.][1]	Uf[.][2]
Start	u^n	v^n	w^n	p^n	$-$ / u^{n-1}	$-$ / v^{n-1}	$-$ / w^{n-1}	$-$ / U_{fx}^{n-1}	$-$ / U_{fy}^{n-1}	$-$ / U_{fz}^{n-1}
end nonLinear	$-$	$-$	$-$	$-$	u^n / u^{n-1}	v^n / v^{n-1}	w^n / w^{n-1}	U_{fx}^n / U_{fx}^{n-1}	U_{fy}^n / U_{fy}^{n-1}	U_{fz}^n / U_{fz}^{n-1}
end waveProp	u^*	v^*	w^*	$-$	u^n / u^{n-1}	v^n / v^{n-1}	w^n / w^{n-1}	U_{fx}^n / U_{fx}^{n-1}	U_{fy}^n / U_{fy}^{n-1}	U_{fz}^n / U_{fz}^{n-1}
start setPForce	u^n	v^n	w^n	$-$	u^* / u^{n-1}	v^* / v^{n-1}	w^* / w^{n-1}	$-$ / U_{fx}^n	$-$ / U_{fy}^n	$-$ / U_{fz}^n
end setPForce	u^n	v^n	w^n	$-$	u^* / u^{n-1}	v^* / v^{n-1}	w^* / w^{n-1}	$\nabla\cdot\mathbf{u}/\Delta t$ / U_{fx}^n	$-$ / U_{fy}^n	$-$ / U_{fz}^n
end Solve (P)	u^n	v^n	w^n	p^{n+1}	u^* / u^{n-1}	v^* / v^{n-1}	w^* / w^{n-1}	$-$ / U_{fx}^n	$-$ / U_{fy}^n	$-$ / U_{fz}^n
end project	u^n	v^n	w^n	p^{n+1}	$-$ / u^{n-1}	$-$ / v^{n-1}	$-$ / w^{n-1}	$\frac{-u^{**}}{\nu\Delta t}$ / U_{fx}^n	$\frac{-v^{**}}{\nu\Delta t}$ / U_{fy}^n	$\frac{-w^{**}}{\nu\Delta t}$ / U_{fz}^n
update velocity storage	u^n	v^n	w^n	p^{n+1}	$-$ / u^{n-1}	$-$ / v^{n-1}	$-$ / w^{n-1}	$\frac{-u^{**}}{\nu\Delta t}$ / U_{fx}^n	$\frac{-v^{**}}{\nu\Delta t}$ / U_{fy}^n	$\frac{-w^{**}}{\nu\Delta t}$ / U_{fz}^n
end solve (U)	u^{n+1}	v^{n+1}	w^{n+1}	p^{n+1}	$-$ / u^n	$-$ / v^n	$-$ / w^n	$-$ / U_{fx}^n	$-$ / U_{fy}^n	$-$ / U_{fz}^n

Figure 5.1: Arrangement of internal storage during timestepping loop (see `dns/integrate.cpp`) for a three-velocity-component, second-order-time, stepping scheme. Presence of a '-' indicates that the relevant storage is free for over-writing if desired. Diagonal lines divide the upper- and lower-order storage so that e.g. the first entry under column labelled `Us[.][0]` corresponds (above diagonal) to `Us[0][0]` and (below) to `Us[1][0]`. The number of levels corresponds to `N_TIME`; i.e. what is shown here corresponds to `N_TIME=2`. For `N_TIME=1` the entries below the diagonals do not exist, while for `N_TIME=3`, there would be an additional level for $n-2$-type entries. If the problem is three-dimensional (`N_Z>1`), field variables are held in the Fourier-transformed state except during computation of nonlinear terms (where products are computed in physical space).

Contents of main directories The following directories are present in the distribution: `include` which has copies of header files; `src` which holds C++ source code for the classes shared by *Semtex* applications; `veclib` which holds routines for many standard loops over vectors, with a mnemonic naming convention (the code here is almost exclusively written in C); `femlib` has one-dimensional spectral polynomial routines (knot and quadrature point computations, differentiation matrix construction) and FFTs, all written in either C or Fortran77; `utility` routines for pre and post-processing; `test` which has code validity regression tests, with the 'right answers' stored in `regress`; `mesh` holds a selection of session files. The two main application code directories are `elliptic` and `dns` which have been described in earlier chapters. The `sm` directory contains useful *SuperMongo* macros, while the `python` directory has some useful scripts for data processing.

The code has a fair amount of low-level optimisation built in for vector computer architectures, but it's not compiled in by default. To get vector-optimised routines, add `-D_VECTOR_ARCH` to the section of `src/Makefile` appropriate to your machine. Also you may want to try altering the parameter `LVR` in `src/temfftd.F` if your job makes heavy use of FFTs.

5.2 Altering the code

(The following discussion assumes you are using the compile-in-source build model, i.e. `make`, rather than compile-out-of-source with `cmake`. However, similar considerations apply in either case.)

If you want to alter the supplied source code, the first thing you need to know is that `make` will seek to resolve source file names in the current directory first, before looking elsewhere (e.g. the `src` directory, the `include` directory). This means that the best strategy is to copy (if not already present) *only* the relevant files which you need to change into your current development directory, typically from the `src` directory, and alter these. This way your new version does not interfere with the supplied code base, and it is also readily apparent exactly which files you have had to change.

For example, say you want to add some new functionality to the `AuxField` class, within the `dns` application. Make a clean copy of the source files (`Makefile`, `*.cpp`, `*.h`) in `dns` to another directory at the same level. In that directory, place copies of `auxfield.cpp` and (if required) `auxfield.h` from `../src` and then work on these.

Testing. Typically when adding code features you want to be sure that you haven't broken existing functionality. The easy way to check is to use the `testregress` script in the `test` directory. It will tell you if the code passes or fails standard regression tests which exercise most code features.

Chapter 6

Utility programs

Part of the strength of *Semtex* lies in the accompanying utility programs that are useful to the CFD analyst. Many of the utilities below will read from standard input and write to standard output, allowing the user to chain operations together in the standard Unix workflow model. At present, all utility programs support serial operation only. Below we describe the purpose and use of compiled utility programs whose source code resides in the `utility` directory. In this directory you will also find a few shell scripts such as `addquick`, `rayavg`, `pline`, `restart`, `run_job` and `save`, while some further *python*-based utilities are included in the `python` directory — none of these utilities are as yet documented below; you may consult the `README` file in the `python` directory for some further guidance on the *python* facilities.

6.1 `addfield`

This utility is used to compute and add more variables derived from the original solution variables.

```
$ addfield -h
Usage: addfield [options] -s session dump.fld
options:
  -h        ... print this message
  -q        ... add kinetic energy per unit mass 0.5(u.u) (default)
  -d        ... add divergence abs(div(u))
  -v        ... add vorticity w=curl(u)
  -e        ... add enstrophy 0.5(w.w)
  -H        ... add helicity 0.5(u.w) (3D only)
  -L        ... add divergence of Lamb vector, div(uxw)
  -g        ... add strain rate magnitude sqrt(2SijSji)
  -D        ... add discriminant of velocity gradient tensor
                NB: divergence is assumed to be zero. (3D only)
  -J        ... add vortex core measure of Jeong & Hussain (3D only)
  -a        ... add all fields derived from velocity (above)
  -f <func> ... add a computed function <func> of x, y, z, t, etc.
  -n        ... turn off mass-matrix smoothing of computed variables
```

The computed variables are added to those in the original field file and a new binary file is output. Here is an example where we add vorticity to the field file produced by running `dns` on the `taylor2` session file produced in § 3.2:

```
$ convert taylor2.fld | head -12
taylor2                   Session
Fri Jan 04 10:44:51 2019  Created
11 11 1 4                 Nr, Ns, Nz, Elements
20                        Step
0.4                       Time
```

```
0.02                      Time step
0.01                      Kinvis
1                         Beta
uvp                       Fields written
ASCII                     Format
 1.745305889e-15  4.584452860e-15  -0.004570994584
-1.028018800e-06    0.09562927785  2.193775364e-14
$ addfield -v -s taylor2 taylor2.fld | convert | head -12
taylor2                   Session
Sat Jan 05 15:05:10 2019  Created
11 11 1 4                 Nr, Ns, Nz, Elements
20                        Step
0.4                       Time
0.02                      Time step
0.01                      Kinvis
1                         Beta
uvpt                      Fields written
ASCII                     Format
 1.745305889e-15  4.584452860e-15  -0.004570994584     5.806132467
-1.028018800e-06    0.09562927785  2.193775364e-14     5.775032679
```

There is only a single component of vorticity added (`t`) since the solution is 2D2C; the (x, y, z) components of vorticity are named (r, s, t).[1] Note also the use made above of the `convert` utility (see §6.5) to convert a binary field file to ASCII for human readability.

There is a point of note when you are using cylindrical coordinates. While, as explained in Blackburn and Sherwin (2004) and Blackburn et al. (2019), primitive variable results computed by `dns` and `elliptic` display spectral convergence at all locations including the coordinate system axis, the same cannot be said of results involving spatial derivatives computed by `addfield`, since it does not employ l'Hopital's rule when taking derivatives at the axis. Usually it will set derivatives to zero on the axis; hence, things such as divergence or vorticity may appear strange there. If the primitive variables look OK (smooth), very likely all is well.

6.2 `calc`

The `calc` utility is a command-line double-precision calculator that uses the same parser and predefined internal variables as `dns` and `elliptic`, and is in concept much like the `yacc-bison` `hoc3` program described by Kernighan and Pike (1984). It is very useful for checking strings that you intend to place in a session file for use by the internal parser. When called with `-h` it lists all the predefined internal variables and functions.

```
$ calc -h
-- Preset internal variables:
N_PROC          = 1
BETA            = 1
KINVIS          = 1
STEP_MAX        = 500
N_TIME          = 2
IO_WSS          = 0
CHKPOINT        = 1
IO_HIS          = 10
CYLINDRICAL     = 0
...
...
SVV_MN          = -1
```

[1] The convention in *Semtex* is that scalar fields take single-character names.

```
N_Z             = 1
ITERATIVE       = 0
C_SMAG          = 0.10000000000000001
SVV_MZ          = -1

-- Calculator operators, functions and procedures:
Unary:      -
Binary:     -, +, *, /, ^ (exponentiation), ~ (atan2), & (hypot), % (fmod)
Functions:  sin,  cos,  tan,  asin,  acos,  atan,
            sinh, cosh, tanh, asinh, acosh, atanh,
            abs, floor, ceil, int,   heav (Heaviside),
            log, log10, exp,  sqrt,  white
            erf, erfc, gamma, lgamma, sgn,
            j0, j1, y0, y1,
Procedures: step, jn, yn, rad, ang, rejn,imjn, jacobi, womcos,womsin
```

Here is a small example:

```
$ calc
x=-2
y=2
DEG*atan(y/x)
-45
DEG*y~x
135
```

Terminate input with `^D` (`EOF`) or `^C` (`SIGINT`).

6.3 `chop`

This is a small but very useful utility for dealing with ASCII text files, often used with the `slit` utility (§ 6.23) to 'slice and dice' columnar data files, though it has a variety of other uses. Probably there is a standard Unix utility that does all of what `chop` does, and more — but `chop` is small and beautiful. All it does is reproduce lines from a text file on a requested numbered basis (see an example in the following section).

```
$ chop -h
usage: chop [options] [input]
  options:
  -h         ... display this message
  -n <lines> ... reproduce this many lines of file    [Default: to EOF]
  -s <line>  ... start at this line number            [Default: 1]
  -S <num>   ... skip <num> lines between each output [Default: 1]
```

6.4 `compare`

The `compare` utility deals with the `<USER>` section of session files, and has two principal modes of operation. If called with just a session file on the command line, it uses the definitions of the `<USER>` section to prepare a field file. This can be useful, for example, for producing files of initial conditions. If called with both a session file and a field file, the information in the `<USER>` section is computed and the data in the field file is subtracted from it; this is useful e.g. when comparing a computed and analytical solution. In this second mode of operation, the difference is written to the standard output as a binary field file, and a summary of the largest difference in each variable is written to standard error — this feature is used by the regression checks in the `test` subdirectory.

```
$ compare -h
usage: compare [options] session [field.file]
options:
  -h ... display this message
  -n ... print 'noise-level' for small errors
  -t ... forward Fourier transform output
```

In the following example, we use compare to generate a laminar initial condition for a channel flow, then add white noise in Fourier mode 1 to initiate transition (see § 6.14); this is is a fairly standard model for (eventually) generating a turbulent flow and is also used in chapter 7, but there with noise in all Fourier modes.

```
$ chop -s 7 -n 6 ../mesh/chan3
<USER>
        u = 1.0-y*y
        v = 0.0
        w = 0.0
        p = 0.0
</USER>
$ compare ../mesh/chan3 | noiz -m 1 -p 1e-3 > chan3.rst
```

6.5 convert

The adopted *Semtex* standard format for field data files is: a 10-line ASCII header, followed by binary data (see § 2.6). But who wants to read binary data? The `convert` utility exists to convert between binary and ASCII data formats. Also, it can convert IEEE-little-endian binary data files to IEEE-big-endian binary data files, which can be useful when moving/processing data produced on one computer to/on another, though *Semtex* codes should normally be able to detect the difference and do the conversion without giving notice. One other useful feature of `convert` is that it can be used to extract a specific field dump from within a sequence stored in a single field file (use `-n` and `-b` together to get the requested dump in binary format).

```
$ convert -h
Usage: convert [-format] [-h] [-v] [-o output] [input[.fld]]
format can be one of:
  -a ... force ASCII output
  -b ... force IEEE-binary output
  -s ... force IEEE-binary output (byte-swapped)
other options are:
  -h        ... print this message
  -n dump   ... select dump number
  -v        ... be verbose
  -o output ... output to named file
  -z        ... zero Time and Step in output
```

In binary data format, field variables are stored sequentially in the order listed in the header. However, when printed out in ASCII format, these different variables appear in sequential columns, as we already saw in § 6.1.

6.6 eneq

From a `.avg` statistics file collected from `dns` with `AVERAGE=3` set, compute terms in either the TKE (turbulent kinetic energy) equation or the MKE (mean kinetic energy) equation. If you are planning to use this utility, this is a case where we must suggest you read the header of the source code (`utility/eneq.cpp`), since there is too much information to reasonably present in this user guide.

```
$ eneq -h
Usage: eneq [options] session dump.avg
options:
  -h ... print this message
  -m ... output terms for mean flow energy instead of TKE
```

6.7 assemble

This utility generates global numberings for element-edge nodes that are used in spectral element assembly operations, and writes an ASCII file (e.g. `session.num`). Depending on the mode of operation, it can produce 'naive' ascending element-by-element numbering, or bandwidth-minimisation numberings of various degrees of exhaustiveness using the Reverse-Cuthill-McKee (RCM) algorithm described e.g. in George and Liu (1981) and also used in `SPARSPAK`, which is admittedly antiquated. Such global numberings are needed for all the elliptic solvers used in *Semtex*. However, this utility is now provided mostly for verification purposes, as the numbering schemes are computed on the fly by the executable that requires them, like `elliptic` and `dns`. It replaces the old utility called `enumerate`, which generated an assembly map or numbering file that the executables read in.

```
$ assemble -h
usage: assemble [options] session
options:
  -h       ... display this message
  -v       ... set verbose output
  -n N     ... override number of element knots to be N
  -O [0-3] ... bandwidth optimization level [Default: 3]
```

One point to note is is that since Dirichlet BC data are lifted out of the Galerkin solution, nodes which correspond to Dirichlet BCs are partitioned to the end of the numbering and are not considered in bandwidth minimisation. Another point is that if the corresponding elliptic problem is singular, e.g. a pressure field with all-Neumann BCs, the solver will select the highest-numbered pressure node to be assigned a homogeneous Dirichlet value (i.e. the solution is pinned to zero at an arbitrary location) in order to make the problem non-singular.

6.8 integral

The `integral` utility approximates the integral of each scalar in a field file over the domain area using Gauss–Lobatto–Legendre quadrature. Also, the area of the domain is similarly approximated. If the session file has `N_Z` > 1 then the values are multiplied by $L_z = 2\pi/\beta$ to give volume integrals. If the coordinate system is cylindrical, the integrals computed further are weighted by radius. The (x, y) centroidal locations of each integral are also reported.

```
$ integral -h
Usage: integral [options] session [dump]
options:
-h ... print this message
-v ... verbose output
-c ... switch cylindrical coordinates off, if defined in session
```

6.9 interp

`Interp` interpolates field data onto a set of (x, y) points supplied as ASCII data. If the session file and field dump are three-dimensional then the two-dimensional interpolation is carried out on every $z-$plane. If the set of data points has a header indicating it is the output of `meshpr` (cf. § 6.12) — which is the standard usage mode — then the interpolated data is output as an ASCII-format field file (with header), otherwise the outcome is also ASCII data but without a header. The basic purpose

of the utility is to facilitate interpolation of data from one domain onto another (the domains do not have to conform but it is assumed that they overlap).

```
$ interp -h
Usage: interp [options] -s session dump
  options:
  -h      ... print this message
  -q      ... run quietly: omit warnings about unlocated points
  -m file ... name file of point data [Default: stdin]
  -v      ... verbose output
```

Each of the points for which interpolation is requested is located in the `session` domain using Newton–Raphson iteration and an element-by-element search; the convergence of the iteration can be controlled by variables `NR_MAX` and `TOL_POS`, which can be reset from their default values in the `session` file. If a requested point cannot be found then a warning is issued (unless the `-q` command-line flag is set) and zero values are returned for the interpolated variables.

The key to understanding its use is to realize that two session files are typically required. The 'to' session file is not used directly by `interp` but is used (by `meshpr`) to generate a set of points at which `interp` will interpolate existing ('from') field file data using a corresponding 'from' session file, e.g.

```
$ meshpr to | interp -q -s from from.fld | convert > to.rst
```

6.10 mapmesh

This utility may be used to reposition x and/or y locations of `NODES` within a session file according to mapping functions supplied on the command line; the output is a new session file. Potentially one could use this with the `rectmesh` utility (§ 6.19) to remap a logically/originally rectangular domain onto a shape that you need.

```
$ mapmesh -h
Usage: mapmesh [options] session
  options:
  -h       ... print this message
  -x <string> ... x <-- f(x, y), f is supplied by string
  -y <string> ... y <-- g(x, y), g is supplied by string
```

6.11 meshplot

`Meshplot` generates a PostScript description of the 2D mesh information generated by `meshpr`. This output can be converted to PDF or displayed on a screen using external utilities such as `ps2pdf` and `gv`. Optionally, if an appropriate viewer is installed, `meshplot` can directly call for display of a named output file. It can read from standard input (e.g. output of `meshpr`) and write to standard output.

```
$ meshplot -h
Usage: meshplot [options] [file]
  options:
  -h        ... display this message
  -a        ... do not show axes
  -i        ... show element-internal mesh
  -n        ... show element numbers
  -o <file> ... write output to named file [Default: stdout]
  -d <prog> ... call prog to display named PostScript output file
  -b 'xmin,xmax,ymin,ymax' ... limit view to region defined by string
```

6.12 meshpr

We often need the mesh locations within elements; this information is provided by `meshpr` ('mesh printer'). The number of points along each element edge (`N_P`) and number of planes (`N_Z`) are taken from the supplied session file but these values can be over-ridden by command line flags.

```
$ meshpr -h
usage: meshpr [options] session
options:
  -h       ... display this message
  -c       ... disable checking of mesh connectivity
  -s       ... list surfaces not determined by mesh connectivity (only)
  -v       ... set verbose output
  -u       ... set uniform spacing [Default: GLL]
  -3       ... produce 3D mesh output: Np*Np*Nz*Nel*(x y z)
  -n <num> ... override number of element knots to be num
  -z <num> ... override number of planes to be num
  -b <num> ... override wavenumber beta to be <num> (3D)
```

```
$ meshpr ../mesh/vb1 | head
11 11 1 15 NR NS NZ NEL
                   0                       0
 0.01319971391838815                       0
 0.04310330526737112                       0
 0.08695293460075898                       0
  0.1408483728826121                       0
  0.2000000000000001                       0
  0.2591516271173879                       0
   0.313047065399241                       0
  0.3568966947326289                       0
```

Note that the output of meshpr is ASCII and has a single-line header that supplies the number of points in each direction in every element (in *Semtex* these are always the same, and here called `NR` and `NS` for historical reasons, instead of `N_P`), followed by the number of z-planes and elements. What follows is a row-major listing of Gauss–Lobatto–Legendre (x, y) mesh locations in every element, i.e. the same ordering as used internally for ordering of data in each z-plane. There is a total of $\mathtt{NR} \times \mathtt{NS} \times \mathtt{NEL}$ such lines.

If the session file is 3D (i.e. $\mathtt{N\_Z} > 1$) that list is followed by $\mathtt{N\_Z} + 1$ lines which provide the coordinates of z-planes (and then one more, for periodic wrapping purposes; the last value is largely irrelevant other than to document the length of the domain in the z coordinate, since the location of the last used z-plane is given by the `N_Z`th value).

6.13 moden

The `moden` utility is used to compute and output the two-dimensional (x, y) distribution of kinetic energy in a particular Fourier mode of a field file. The `-z` command-line flag asks that two z-planes of data be taken as the 0th Fourier mode, e.g. in the case where we are dealing with a single complex eigenmode; otherwise, the kinetic energy in mode 0 is computed only from the real part of the Fourier transform of the input data.

```
$ moden -h
moden [-h] [-m mode] [-z] [input[.fld]
```

6.14 noiz

The `noiz` utility can be used to add Gaussian-distributed white noise of specified standard deviation to all scalars in a field file. If requested, this noise can be added only to specified Fourier modes,

otherwise it is added to all modes/data planes. One minor point to note here with respect to restart files for DNS is that such white-noise perturbations are not divergence-free, though they will become so (or approximately so) after time-stepping commences.

```
$ noiz -h
usage: noiz [options] [input[.fld]]
options:
-h         ... print this help message
-f         ... filter instead of perturb
-o output  ... write to named file
-p perturb ... standard deviation of perturbation  [Default: 0.0]
-m mode    ... add noise only to this Fourier mode [Default: all modes]
-s seed    ... set random number seed              [Default: 0]
```

A lesser-used functionality of `noiz` is, if the `-f` command-line flag is set, to filter out (zero) data of a field file in a single Fourier mode as given on the command line.

6.15 `probe`

This utility provides similar functionality to `interp` (§ 6.9) in that it outputs field data as interpolated onto specified points. However, `probe` differs in that the points may be supplied in three-dimensional space, whereas for `interp` the points are in two-dimensional space, i.e. `probe` also carries out Fourier series interpolation in the z-direction. The `interp` utility is designed mainly to interpolate from one 2D *Semtex* mesh to another, and is capable of writing out an (ASCII-format) field file, while `probe` isn't.

```
$ probe -h
Usage: probe [options] -s session dump
  options:
  -h      ... print this message
  -m      ... minimal output
  -p file ... name file of point data [Default: stdin]
```

Example, with probe point entered on command line (terminated with `EOF/^D`):

```
$ probe -m -s vb1 vb1.fld
0.5 0.5 0.0
      0.023527766      0.005897312      0.078899172      0.0032106831
```

These are the values of (u, v, w, p) at $(0.5, 0.5, 0)$ for the vortex breakdown solution of § 3.4. Without the `-m` flag, probe will also give an integer index of the input point number and echo the location of the (x, y, z) location of each probe point.

6.16 `probeline`

`Probeline` actually just provides an alternative interface to the `probe` utility. Instead of reading the required points from input, it computes them along a line that is specified on the command line as a starting point (x_0, y_0, z_0) and a vector $(\Delta x, \Delta y, \Delta z)$.

```
$ probeline -h
Usage: probeline [-h] -p "[n:]x0,y0,z0,dx,dy,dz" -s session dump
```

The default number of points (`n`) is 64. They are uniformly spaced along the line (like Matlab's `linspace`).

6.17 probeplane

Like probeline, probeplane provides an alternative interface to the probe utility. Instead of internally computing points along a straight line, however, it computes them on a rectangular cutting plane that is orthogonal to the x, y or z direction.

```
$ probeplane -h
Usage: probeplane [options] -s session dump
options:
-h                ... print this message
-xy "x0,y0,dx,dy" ... xy-cutting plane
-xz "x0,z0,dx,dz" ... xz-cutting plane
-yz "y0,z0,dy,dz" ... yz-cutting plane
-orig #           ... origin of the cutting plane along ortho axis
-nx #             ... resolution along the x-axis
-ny #             ... resolution along the y-axis
-swap             ... swap output x <--> y (rotate)
-tec              ... write TECPLOT-formatted ASCII output
-sm               ... write SM-formatted binary output
-0                ... output zero if point is outside mesh (instead of warning)
```

The default number of points (nx or ny) is 64.

6.18 project

The purpose of this utility is to use inbuilt basis functions to project a solution from one polynomial order to another (higher or lower), retaining the same spectral element/Fourier structure. The outcome could then be used e.g. to restart a simulation on the same mesh but at a higher or lower resolution based on p-refinement/coarsening (here p is polynomial order, not pressure). This purpose is distinct from that of the interp utility, which can be used to transfer data from one domain/spectral element mesh (session file) to another. The input file should be a binary field file.

```
$ project -h
Usage: project [options] [file]
  options:
  -h       ... print this message
  -n <num> ... 2D projection onto num X num
  -z <num> ... 3D projection onto num planes
  -w       ... Retain w components in 3D-->2D proj'n [Default: delete]
  -u       ... project to uniform grid from GLL
  -U       ... project from uniform grid to GLL
```

The use of the -n command-line flag to project from one spectral element order to another in (x, y) planes is perhaps both the most common use of this utility and fairly obvious. Note that projecting a three-dimensional solution to a single z-plane (-z 1) effectively takes the spanwise/azimuthal average of the solution, and that in this case, by default, the third velocity component (w) will also be deleted: use the -w command line flag to retain this component if you need to keep it. The operation below will take a spanwise average of a 3D3C data file and keep all velocity components:

```
$ project -z 1 -w chan3.fld | project -z 80 > chan3_mean_in_z.fld
```

6.19 rectmesh

While by design, *Semtex* deals with unstructured (though conforming) meshes, in a surprising number of cases a simple rectangular (x, y) mesh is enough. The rectmesh utility is designed to help you get started in such cases by producing a prototype session file with at least the NODES and ELEMENTS sections as required, leaving you to fix up the other sections using a text editor. The ASCII input

for rectmesh is simple: a list of x-locations, one per line, a blank one-line separator, followed by a list of y-locations. The spectral element mesh is produced by taking the tensor product of these two lists. We direct the reader to § 7.2 for an example.

```
$ rectmesh -h
Usage: rectmesh [options] [file]
  options:
  -h       ... print this message
  -b <num> ... output in <num> blocks, contiguous in x [Default: 1]
  -e <num> ... offset first element number by <num>
  -v <num> ... offset first vertex number by <num>
```

6.20 rstress

This utility has two different modes of use. The name `rstress` originally came about as a contraction of 'Reynolds stress' which was also the original purpose of the utility: given a (single) `.avg` file with accumulated running averages of velocities and their products, e.g. $\langle u \rangle$, $\langle uv \rangle$ obtained by running a simulation with `AVERAGE=2` set in the `TOKENS` section, `rstress` computes covariances by subtracting the products of means from means of products, so that e.g. $\langle uv \rangle \to \langle u'v' \rangle$.

The other mode of operation, given two input files, is to perform binary operations ($-$, $+$, $\times$, $/$) on the pair and write out the outcome. The ordering (where it is significant for $-$ and $/$) is second.op.first. The main use of this mode is to subtract one file (e.g. an analytical solution) from another (e.g. a computed result).

```
$ rstress -h
usage: rstress [options] avg.file [field.file]
options:
  -h         ... display this message
  -<s,a,m,d> ... binary op: subtract, add, multiply, divide [Default: subtract]
```

Please see also the discussions of §§ 4.9 and 7.5.

6.21 sem2tec

This is the primary utility used to prepare *Semtex* output for post-processing in the form of input files for AMTEC's *Tecplot* program. While *Tecplot* is a program which requires you to pay for a licence, there are several open-source programs (*Paraview* and *VisIt*) which will read *Tecplot* input files (here with a `.plt` extension). To speed up the process, `sem2tec` makes a system call to `preplot`, an AMTEC-supplied utility, to convert the output of `sem2tec` from ASCII to binary.

```
$ sem2tec -h
usage: sem2tec [options] session[.fld]
options:
-h       ... print this message
-o file  ... write output to the named file instead of running preplot
-m file  ... read the mesh from the named file (instead of stdin)
-d <num> ... extract dump <num> from file
-n <num> ... evaluate the solution on an evenly-spaced mesh with N X N
             points.  If N = 0, then no interpolation is done, i.e., the
             output mesh will be on a standard GLL-spectral element mesh
-w       ... extend the data by one additional plane in the z-direction
```

Two points to note initially: (a) each spectral–Fourier element becomes what *Tecplot* calls a 'zone'; (b) by default, `sem2tec` interpolates the solution from the underlying Gauss–Lobatto–Legendre mesh to an isoparametrically-mapped uniform mesh (this helps make smooth solutions seem more smooth to the human eye) — but occasionally you will wish to see the results computed by *Semtex* on

the original Gauss–Lobatto–Legendre mesh, in which case use the `-n 0` command-line instruction. Finally, for three-dimensional solutions, especially in cylindrical coordinates, you may wish to use the `-w` flag to 'wrap' the outcome one more z-plane so that the visualisation appears more periodic.

While we are on the topic of cylindrical-coordinate solutions and *Tecplot*: it can initially seem confusing that the visualisation you will see for three-dimensional solutions appears as though it is in Cartesian space. That's because *Tecplot* doesn't know that the results are in cylindrical coordinates. You have to use equations to map your data into Cartesian coordinates (under `data->alter`). Below we show an example equation file that you can read into *Tecplot* to do the mapping. This also maps velocity components into Cartesian space, though typically you might wish to view (say) cylindrical velocity components (e.g. swirl/w) but in Cartesian coordinates.

```
#!MC 1120
# Convert cylindrical coordinate data to Cartesian.
$!ALTERDATA
  EQUATION="{YC} = {Y}*cos({Z})"
$!ALTERDATA
  EQUATION="{ZC} = {Y}*sin({Z})"
$!ALTERDATA
  EQUATION="{VC} = {V}*cos({Z})-{W}*sin({Z})"
$!ALTERDATA
  EQUATION="{WC} = {W}*cos({Z})+{V}*sin({Z})"
```

6.22 `sem2vtk`

Deprecated. That's because many VTK-based visualisation programs (e.g. *VisIt* and *Paraview*) now seem capable of reading *Tecplot* input files. It lives on just in case you might need to use some other visualisation tool that cannot, and requires `.vtk` files.

By the way: if you just need to visualise three-dimensional isosurfaces, you may find that the `sview` program that is available with *Semtex* is, after some habituation, very much simpler and quicker to use than any of the larger tools such as *Tecplot*. Also, `sview` can read simple input scripts so is suited to producing sequences of graphics files suitable for making animations via a shell-based loop. Its visualisation is based on OpenGL and GLUT.

6.23 `slit`

While the `chop` utility (§ 6.3) reproduces specific rows or ASCII input, `slit` reproduces specific columns. Often the two utilities are used in concert. Unusually, `slit` has no usage prompt to output via a `-h` command-line argument. From the header of `utility/slit.c`:

```
Usage: slit [-c <colstr>] [file], where <colstr> is a
comma-separated list of column numbers.
```

Example, to extract time-series of Fourier mode 2 out of a `.mdl` file (§,4.7) from the outcome of a simulation with 24 z-planes (12 Fourier modes):

```
chop -s 5 -S 12 chan3.mdl | slit -c 1,3 > mode2.dat
```

6.24 `traction`

See also § 4.6. This utility is designed as a post-processing tool that will compute normal (pressure) and tangential (viscous) traction distributions on boundary `SURFACES` that are in the `wall` group from a Navier–Stokes field file.[2] If token `IO_WSS` was set, such computations would have been carried out during runtime, as reported in § 4.6. The output is another (binary) field file but of reduced

[2] At a solid wall in incompressible flow, wall-normal viscous traction is zero.

dimensionality: if the original solution domain was two-dimensional, the outcome is one-dimensional; if it was three-dimensional the outcome is two-dimensional: this reduction occurs because walls are one dimension lower than domains. The output variables are called (n, t) (2D) or (n, t, s) (3D): n being the normal traction (i.e. pressure) and t being the tangential viscous traction component that lies in the (x, y) plane; s is the out-of-plane component of viscous traction. Together with the `wallmesh` utility (§ 6.26), one can run `sem2tec` to produce a *Tecplot* input file for visualisation purposes.

```
$ traction -h
Usage: traction [options] session [file]  [options]:
  -h       ... print this message
  -v[vv..] ... increase verbosity level
```

6.25 `transform`

`Transform` is used to carry out a two-dimensional discrete polynomial transform (DPT, Boyd; 2001; Blackburn and Schmidt; 2003) in (x, y) space or a one-dimensional discrete Fourier transform (DFT/FFT) in the z-direction on a field file. The default DPT is to the 'modal' basis, see Karniadakis and Sherwin (2005) or Blackburn and Schmidt (2003).

```
$ transform -h
Usage: transform [options] [file]
options:
-h ... print this message
-P ... Discrete Polynomial Transform (2D)
-F ... Discrete Fourier    Transform (1D)
-B ... do both DPT & DFT [Default]
-i ... carry out inverse transform instead
-l ... use Legendre basis functions instead of modal expansions
```

6.26 `wallmesh`

The `wallmesh` utility extracts and prints up mesh locations for `SURFACES` that are in the `wall` group — it is a postprocessor for `meshpr` that just supplies wall nodal locations, and the format is very similar to what `meshpr` supplies, but of reduced dimensionality (see § 6.24). Its output may be supplied to `sem2tec` in combination with a wall traction file to produce a distribution of wall tractions that can be visualised in *Tecplot*.

```
$ wallmesh -h
usage: wallmesh [options] session [mesh.file]
options:
  -h ... display this message
```

Note that meshpr is first used to supply locations from which only those in the `wall` group are reproduced, and that in the header, `NR=1`, as follows from the reduction in dimensionality, as shown below.

```
$ meshpr chan3 | wallmesh chan3 | chop -n 3
7 1 24 8 NR NS NZ NEL
  -3.141592653589793                    -1
  -3.008250813538205                    -1
```

6.27 `xplane`

The `xplane` utility extracts either one, or optionally two sequential, planes of data from a field file, and outputs a new field file containing just one (or two) planes of data.

```
$ xplane -h
xplane [-h] [-n plane] [-2] [input[.fld]
```

Why, optionally, two sequential planes? If the data have previously been Fourier transformed using `transform -F`, that can pull out a complex mode, with the real part as the first extracted plane, and the imaginary part as the second.

Chapter 7

DNS 101 — Turbulent channel flow

Turbulent Poiseuille flow between two parallel plates is a canonical test case for direct numerical simulation (DNS) codes. While *Semtex* will of course deal with more complicated problems, we'll use this as an example to illustrate techniques of mesh design, and of obtaining transition and extracting turbulence statistics. Our basis for comparison will be the DNS results of Kim, Moin and Moser (1987), obtained with a Fourier–Fourier–Chebyshev code.

A schematic of the configuration is shown with figure 7.1. The flow is assumed periodic in the x (streamwise) and z (spanwise) directions. In the y-direction, non-slip Dirichlet boundary conditions are applied for all velocity components at the upper and lower walls, where also a high-order pressure boundary condition (of computed Neumann type, see Karniadakis et al.; 1991) is employed. Since we are working with a spectral element–Fourier code, we are free to choose either of the x or z directions as the Fourier direction; here, we will use Fourier expansions in the z direction, have a spectral element mesh in the x–y plane, and set up explicit periodicity in the x direction.

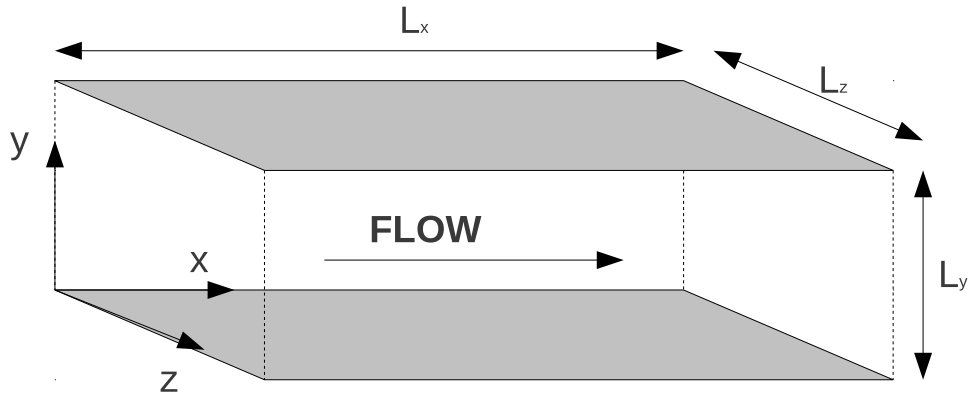


Figure 7.1: Channel flow geometry.

7.1 Parameters

To match Kim et al. (1987) we will aim for a bulk flow Reynolds number based on the centreline mean speed U and channel half-height $\delta = L_y/2$ of $Re_\delta = U\delta/\nu = 3300$. For this flow the associated Reynolds number based on the friction velocity $u_\tau = (\tau_w/\rho)^{1/2}$ and half-height is $Re_\tau = u_\tau\delta/\nu = 183$. (It is worth noting that the relationship between the bulk and friction Reynolds numbers for turbulent flows is empirically based. For channel flow, a reasonable approximation is given by $Re_\delta/Re_\tau = 2.5\ln Re_\tau + 5$, while for turbulent flow in a pipe of diameter D, Blasius' correlation $Re_\tau = u_\tau D/2\nu = 99.44\times10^{-3}Re_D^{7/8}$ is quite good at moderate Reynolds numbers.)

Kim et al. (1987) used domain extents of $L_x = 4\pi\delta$ and $L_z = 2\pi\delta$ but for illustrative purposes we will choose the smaller sizes $L_x = 2\pi\delta$ and $L_z = \pi\delta$. We will find that the turbulence statistics examined are apparently little affected by this reduction in domain size, but the cost of simulation is significantly reduced. Since we will use Fourier expansions in the z direction we will have the basic spanwise wavenumber $\beta = 2\pi/L_z = 2$. This is set below by the simulation token `BETA = 2`.

We will choose second-order time integration, usually the best compromise between stability and accuracy. (This is set below by the token `N_TIME = 2`, which could actually be omitted from the session file since that is the default value.)

We need to choose a spectral element polynomial order. Values around 9 provide a good compromise between speed and accuracy for this code. This is set with the token `N_P = 10` (the number of points along the edge of an element, one more than the polynomial order). Sometimes in what follows we will also approximate the typical number of mesh divisions along the the edge of an element as 10, although the sharp-eyed reader will note that it should logically be 9.

We need to choose the kinematic viscosity, ν. We aim to have $Re_\delta = 3300$. It is good simulation/numerical practice to aim for a characteristic velocity scale of unity, as well as a characteristic mesh length scale of unity. In this case these goals imply that we aim for a centreline mean speed $U \simeq 1$ and our channel half-height $\delta = 1$. With these choices we are left with

$$\nu = Re_\delta^{-1} = 303 \times 10^{-6},$$

which is set by the simulation token `KINVIS = 303e-6`.

When using periodicity in the streamwise direction, a body force is required in order to maintain the flow. This is needed because the pressure as well as the velocity is required to be streamwise-periodic. The required body force per unit mass (i.e. acceleration) is calculated from a time-average force balance in x-direction for the entire channel

$$\begin{aligned} \rho \times f_x \times \underbrace{\cancel{L_x} \times L_y \times \cancel{L_z}}_{\text{channel volume}} &= 2 \times \tau_w \times \underbrace{\cancel{L_x} \times \cancel{L_z}}_{\text{one wall surface}} \\ \rho \times f_x \times \delta &= \tau_w \\ \frac{f_x \delta}{U^2} &= \left(\frac{u_\tau}{U}\right)^2 = \left(\frac{Re_\tau}{Re_\delta}\right)^2 \end{aligned}$$

where τ_w is the time-average wall shear stress. With $\rho = \delta = U = 1$ and (from correlation/previous results) $Re_\tau = 183$, $Re_\delta = 3300$, the required value is

$$f_x = 3.08 \times 10^{-3}.$$

This will be set with the token `CONST_X = 3.08e-3` in the `<FORCE>` section. Note that if required for body forces in the y or z directions there are corresponding tokens `CONST_Y` and `CONST_Z` respectively (see section 4.13 for other types of forcing). These body forces are added to the component momentum equations (the Navier–Stokes equations). Also note that the code is written assuming $\rho = 1$.

The friction velocity $u_\tau = (\tau_w/\rho)^{1/2} \equiv U Re_\tau / Re_\delta = 55.5 \times 10^{-3}$, and the viscous wall length scale $l_w = \nu/u_\tau = 5.46 \times 10^{-3}$.

7.2 Mesh design

In designing the mesh for the channel flow, rules of thumb established in related studies (Piomelli; 1997; Kim et al.; 1987; Blackburn and Schmidt; 2003) have been considered. All mentioned coordinates are with respect to coordinate system established in this work. Thus, x is the streamwise direction, y the wall-normal and z the spanwise direction — Fourier expansions are always used in the z direction.

Kim et al. (1987) used $\Delta x^+ = \Delta x / l_w = 12$, $y^+ = 0.05$ (at the wall) and $\Delta z^+ = 7$ in their study. (Piomelli; 1997) proposed similar values with $\Delta x^+ = 15$, $\Delta y^+ < 1|_{\text{wall}}$ and $\Delta z^+ = 6$.

The wall-normal part of the spectral element mesh design strategy for wall-resolved LES described by Blackburn and Schmidt (2003) is to terminate the element closest to the wall at $y^+ = 10$, the second element at $y^+ \simeq 35$, then use a geometric progression of sizes to reach the flow centreline (one has to choose the number of elements and geometric expansion factor). This ensures there is good resolution in the viscous-dominated wall layer as well as the buffer layer ($10 < y^+ < 35$), where turbulent energy production is greatest. Nevertheless, here, the second element layer is reduced to $y^+ = 25$ in order to improve resolution in the middle of the buffer layer.

The first two element heights in the wall-normal direction are then $\Delta y_1(10) = 0.056$ and $\Delta y_2(25) = 0.139$. For the remainder, we use a geometric progression of four elements starting with an initial height of $\Delta y = 0.139 - 0.056 = 0.083$ to reach the channel centreline. The total number of elements in the y-direction is 12.

Next considering the x-direction, the indicative mesh spacing is $\Delta x^+ = 15$, corresponding to a length $\Delta x = 15 \times 5.46 \times 10^{-3} = 81.9 \times 10^{-3}$. The number of grid points needed to cover the domain extent in the x-direction is then of order $N_x = 2\pi/\Delta x = 76.7$. For a mesh with `N_P=10` we need of order 8 elements. So our (x, y) spectral element mesh is now of size $8 \times 12 = 96$ elements.

Finally considering the z direction, we have $L_z = \pi$ and want $\Delta z^+ \simeq 7$, or $\Delta z \simeq 7 \times 5.46 \times 10^{-3} = 38.2 \times 10^{-3}$. The implied number of z planes is then of order $\pi / 38.2 \times 10^{-3} = 82.2$. We need an integer number of planes which must be even (one prime factor of 2) and other allowable prime factors of 3 and 5. 80 planes seems convenient so we choose `N_Z = 10`. Note that means we could run on either a single processor in serial execution or 2, 4, 8, 10, 20 or 40 in parallel. There will be 40 Fourier modes.

The mesh can be created using the provided tool `rectmesh` which reads from an input file containing two blocks with all grid lines in x and y separated by an empty line. The output from running `rectmesh` will be a valid *Semtex* session file but which will generally need some editing.

```
rectmesh [options] mesh.inp > sessionfile
```

A `rectmesh` input file for this case is as follows:

```
-3.14159265358979
-2.35619449019234
-1.5707963267949
-0.785398163397448
0.0000
0.785398163397448
1.5707963267949
2.35619449019234
3.14159265358979

-1.0
-0.944
-0.861
-0.745
-0.575
-0.330
0.0
0.330
0.575
0.745
0.861
0.944
1.0
```

Once this file is built, all the above mentioned parameters can be changed to their desired value. Also, the boundary conditions have to be named and set into place. We use wall boundary conditions on the upper and lower edges of the domain, and edit the <SURFACES> section to obtain periodicity in the streamwise direction. Here is the example input file for this particular case (which is much the same as the chan3 session file in the mesh directory):

```
#####################################################################
#  96 element channel flow, for Re_bulk = 3300, Re_tau = 183

<FIELDS>
 u v w p
</FIELDS>

<USER>
    u = 1.0-y*y
    v = 0.0
    w = 0.0
    p = 0.0
</USER>

<TOKENS>
    N_TIME  = 2
    N_P     = 10
    N_Z     = 80
    BETA    = 2.0
    D_T     = 0.01
    T_FINAL = 800
    N_STEP  = int(T_FINAL/D_T)
    KINVIS  = 303e-6
    IO_CFL  = 100
    IO_HIS  = 100
    AVERAGE = 2
</TOKENS>

<FORCE>
    CONST_X = 3.08e-3
</FORCE>

<GROUPS NUMBER=1>
    1 w wall
</GROUPS>

<BCS NUMBER=1>
    1 w 4
        <D> u = 0.0 </D>
        <D> v = 0.0 </D>
        <D> w = 0.0 </D>
        <H> p       </H>
</BCS>

<NODES NUMBER=117>
    1        -3.14159             -1              0
  ...
  117         3.14159              1              0
</NODES>

<ELEMENTS NUMBER=96>
    1 <Q>    1    2   11   10    </Q>
```

```
  ...
   96 <Q>  107  108  117  116    </Q>
</ELEMENTS>

<SURFACES NUMBER=28>
    1    1    1    <B> w </B>
    2    2    1    <B> w </B>
    3    3    1    <B> w </B>
    4    4    1    <B> w </B>
    5    5    1    <B> w </B>
    6    6    1    <B> w </B>
    7    7    1    <B> w </B>
    8    8    1    <B> w </B>
    9   89    3    <B> w </B>
   10   90    3    <B> w </B>
   11   91    3    <B> w </B>
   12   92    3    <B> w </B>
   13   93    3    <B> w </B>
   14   94    3    <B> w </B>
   15   95    3    <B> w </B>
   16   96    3    <B> w </B>
   17    8    2    <P>   1      4       </P>
   18   16    2    <P>   9      4       </P>
   19   24    2    <P>  17      4       </P>
   20   32    2    <P>  25      4       </P>
   21   40    2    <P>  33      4       </P>
   22   48    2    <P>  41      4       </P>
   23   56    2    <P>  49      4       </P>
   24   64    2    <P>  57      4       </P>
   25   72    2    <P>  65      4       </P>
   26   80    2    <P>  73      4       </P>
   27   88    2    <P>  81      4       </P>
   28   96    2    <P>  89      4       </P>
</SURFACES>

<HISTORY NUMBER=1>
 1 0.0 0.0 0.0
</HISTORY>
```

7.3 Initiating and monitoring transition

The critical bulk Reynolds number for linear instability of channel flow is $Re_c = 5772$ but turbulence can typically be maintained down to lower values (and here we are aiming at $Re_\delta = 3300$) if transition is obtained. Our strategy here will be to start from a laminar flow profile and add some white noise to obtain transition. This is done using the following single line to generate an initial condition:

```
compare chan | noiz -p 0.1 > chan.rst
```

Here, the command-line value 0.1 represents the standard deviation of the normal/Gaussian distribution from which the noise is derived using a pseudorandom number generator and is therefore quite large compared to an average velocity of $U = 1$. It is necessary, since our Reynolds number is here below the critical value, to assure a sufficient amount of disturbance to initiate transition to a turbulent state. If the Reynolds number were above the critical value, any perturbation level above machine noise level should eventually produce transition — it just depends on how long you are prepared to wait.

We note that channel flow is a case for which it is relatively easy to obtain transition to turbulence without carefully manipulating the noise level, or restricting the time step. In other flows (pipe flow

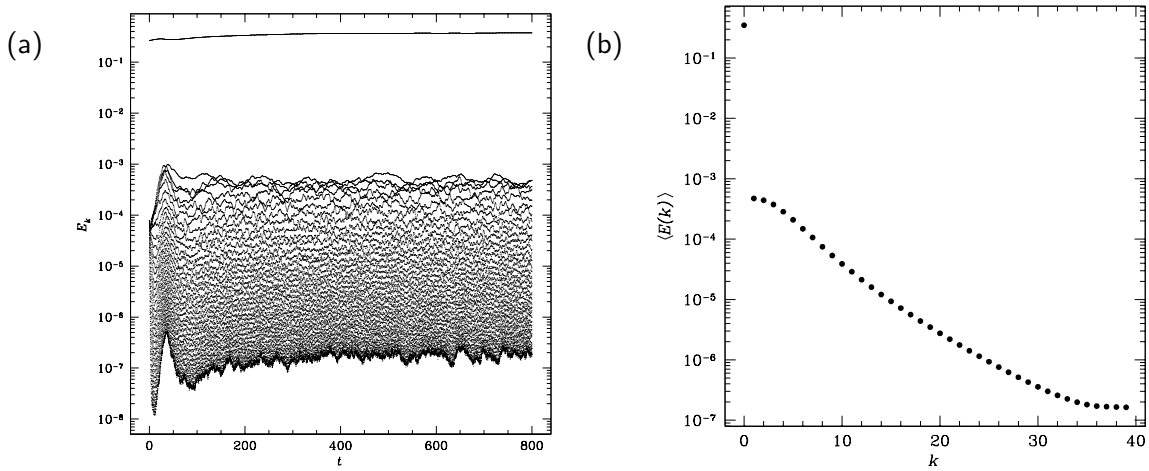


Figure 7.2: Modal energies for (a) the individual wave numbers over time and (b) by wavenumber. These images were prepared using *SuperMongo* macros.

being a notorious example) transition can be quite a violent process and it can be hard to nurse a simulation through it to reach a turbulent, though ultimately more energetically benign, state. Expedients one can try are to (*a*) introduce the initial perturbation only in Fourier mode 1 rather than all modes, as was done in the example above (see § 6.14), (*b*) reduce Reynolds number, (*c*) add spectral vanishing viscosity (see § 4.12) to contain energy in high wavenumbers during transition, (*d*) increase spatial resolution, (*e*) reduce timestepping order. Once transition completes, it may be possible to revert to the original settings. Another common option is to project, interpolate or otherwise reuse an existing turbulent restart file from another simulation; if one is just changing Reynolds number, domain length, or perhaps viscosity model, this is the recommended approach — though obviously one needs an existing outcome in a somewhat similar geometry to start from.

In order to monitor transition, a very useful diagnostic is to monitor the evolution of kinetic energies in all Fourier modes. Transition should be easy to see as a moderately rapid increase in turbulent energy within small scales that are represented by higher wave numbers. Figure 7.2(a) shows the transition for the mentioned channel flow. Figure 7.2(b) indicates that the resolution is adequate due to a separation of about three and a half decades between the highest and smallest wave number. An increase of energy within the highest wave numbers would also indicate an under-representation because of aliasing of the energy of higher modes back into the resolved frequency domain (which is the underlying reason for the plateau seen at the highest wavenumbers in figure 7.2(b)).

Figure 7.2(a) shows the evolution of kinetic energies in the 40 Fourier modes represented. By way of interpretation, the mode with highest energy (here, and typically) is mode 0, which represents the two-dimensional or z-average flow. All the other modes start off with much the same energy, which is a result of having chosen to pollute all modes equally when using the `noiz` utility (often, one would just pollute mode 1 and allow convolution to distribute energy to all other modes — this gives a more gentle perturbation). The highest modes typically decay rather rapidly initially, with the lower modes either slowly losing or (as here, gaining) energy. Also the energy in mode 0 here increases a little over time, partly because the initial condition here actually had a lower volumetric flow rate than the equilibrium turbulent flow. At $t \approx 70$ the flow makes a transition to a turbulent state, typically signalled by the higher modes gaining energy fairly rapidly until a quasi-equilibrium is reached. Eventually the flow settles to a statistical equilibrium at $t \approx 600$. Figure 7.2(b) shows the temporal average values of energies in the various Fourier modes. (The *SuperMongo* macros `moden` and `modav` were used to plot figure 7.2.)

At the end of this simulation, the flow should be in an approximately statistical stationary state. Check the x-component tractive force given at the end of `chan.flx` (which should be around 0.0388).

This is the tractive force per unit domain width in the z-direction (see § 4.5). That ought to closely balance the total body force per unit width, which is $f_x \times 2 \times 2\pi = 303 \times 10^{-6} \times 2 \times 2\pi = 0.0387$. Since the flow is turbulent and driven by a constant body force, both the bulk velocity (volumetric flow rate per unit area) and wall tractions will fluctuate somewhat in time, and we should really compare the time-average tractive force with the body force, but evidently the single-time outcome is quite close. The agreement should also depend on the spatial and temporal resolution of the simulation, but the resolution suggested in the session file is quite good for the Reynolds number employed.

7.4 Timestepping order, restarting, and parallel execution

Semtex's `dns` can run with first, second, or third-order accurate timestepping according the token `N_TIME`, whose default value of 2 implies second-order timestepping, which is generally the best choice. However, CFL stability declines somewhat as timestepping order increases, and if in any case you know that you are headed towards a steady-state outcome, `N_TIME=1` might be a better choice so that comparatively larger timesteps could be used while still maintaining CFL stability. These choices also have a subtle interaction with restarting (which can be accomplished by copying or moving `session.fld` to `session.rst` and running again), since only a single field dump is read from `session.rst`, meaning that the first subsequent timestep can only be first-order accurate (the second can be second-order accurate, etc., up to the order you are running with). For a turbulent flow this effect has only minor overall significance, but it can lead to small disruptions at every restart for a steady or periodic flow; generally it is an effect to be aware of rather than concerned about.

As will be pointed out immediately below, when restarting with token `AVERAGE` > 0, `dns` will also attempt to read in `session.avg` in order to continue accumulating statistics.

7.5 Flow statistics

Flow statistics can be collected by setting the `AVERAGE` token to non-zero values 1, 2 or 3 (here a value of 2 was used, see below). See also § 4.9. To obtain statistical convergence it is necessary to average over a sufficiently long time, just as would be the case in a physical experiment. Statistics are updated every `IO_HIS` simulation steps (here, every 100 steps). In the present case, the total averaging time was chosen to be 400, which represents of order 60 'wash-through' times, since the domain length is 2π and the bulk flow speed $U = 1$. Since the time between data updates is $100 \times 0.002 = 0.2$ there are a total of $400/0.2 = 2000$ averaging buffer updates.

Once the simulation is run, a `.avg` file is produced. For `AVERAGE=1`, statistics for the represented fields are collected, i.e. $\langle u \rangle$, $\langle v \rangle$, $\ldots$, $\langle p \rangle$. For `AVERAGE=2`, averages of velocity field products are stored too, i.e. $\langle uu \rangle$, $\langle uv \rangle$, etc. For `AVERAGE=3`, additional products are collected to allow computation of terms in the fluctuating energy equation. Note that it is necessary to calculate Reynolds stresses and energy equation terms in post-processing (e.g. $\langle u'v' \rangle = \langle uv \rangle - \langle u \rangle \langle v \rangle$). Note also that if `.avg` files exist, they are read in at start of execution of `dns` to initiate averaging buffers: the `Step` value in the file's header stores the number of averages obtained to date.

Having collected statistics, our next step is to calculate the Reynolds stresses and to average in x- and z-direction. To illustrate the possible processing, here is an example shell script:

```
#!/bin/bash

# temporary files
FTNZ='/tmp/chan_nz1'
FTRS='/tmp/reynolds-stress.xy'
FRSY='./reynolds-stress.y'
```

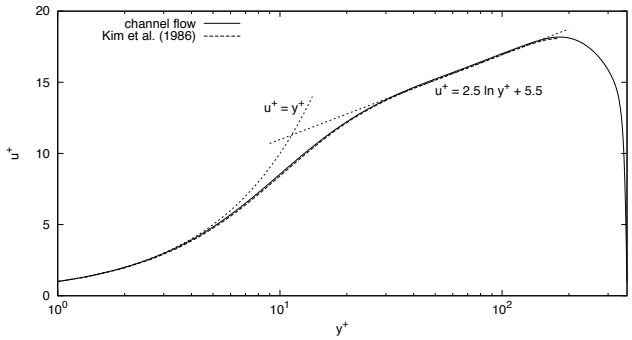


Figure 7.3: Mean velocity profile of the channel flow compared to the law of the wall and data from Kim et al. (1987)

```
# average field results in z
project -z1 chan.avg > /tmp/chan.avg.xy

# calculate reynolds-stresses (xy-plane)
rstress /tmp/chan.avg.xy > $FTRS

# new input file with NZ=1
LNZ=$(cat chan | grep N_Z -n | awk -F: '{print $1}')
sed '$LNZs/.*/  N_Z    = 1/' <chan >$FTNZ

# average reynolds-stresses in x
rayavg npts navg y_0 x_0   y_0 x_1 dy dx $FTNZ $FTRS >  $FRSY
rayavg npts navg y_0 x_1   y_0 x_2 dy dx $FTNZ $FTRS >> $FRSY
...
rayavg npts navg y_0 x_n-1 y_0 x_n dy dx $FTNZ $FTRS >> $FRSY

# PLOTTING -- make a gnuplot script if you like:
gnuplot ~/scripts/plot_rstress.gpl;
```

Figure 7.3 shows the mean velocity profile for the channel flow in comparison to data from Kim et al. (1987). The dashed lines are representing the linear respectively logarithmic part of the law of the wall. Figure 7.4 show rms Reynolds stress values in friction velocity units with comparison to data from Kim et al. (1987). Quite good agreement is evident.

Finally (though it is very important), we should note that one could run this simulation either on a single processor, or, using MPI, on up to 40 processors (and perhaps have sped up execution by up to a factor of 40!). Please see the discussion of § 3.9.2 for further information.

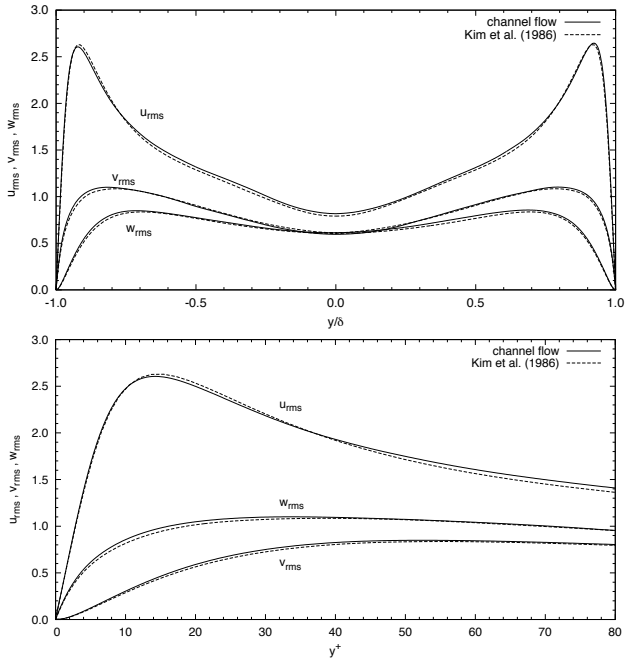


Figure 7.4: Root-mean-squares velocity fluctuations in global coordinates normalised by the wall shear velocity u_τ. Comparison to data from Kim et al. (1987).

References

Åkervik, E., Brandt, L., Henningson, D. S., Hœpfner, J., Marxen, O. and Schlatter, P. (2006). Steady solutions to the Navier–Stokes equations by selective frequency damping, *Phys. Fluids* **18**: 068102–1–4.

Albrecht, T., Blackburn, H. M., Lopez, J. M., Manasseh, R. and Meunier, P. (2015). Triadic resonances in precessing rapidly rotating cylinder flows, *J. Fluid Mech.* **778**: R–1–11.

Amon, C. H. and Patera, A. T. (1989). Numerical calculation of stable three-dimensional tertiary states in grooved-channel flow, *Phys. Fluids A* **1**(2): 2005–2009.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D. (1999). *LAPACK User's Guide*, 3rd edn, SIAM.

Barrett, R., Berry, M., Chan, T. F., Demmell, J., Donato, J. M., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd edn, SIAM.

Batchelor, G. K. (1967). *An Introduction to Fluid Dynamics*, Cambridge University Press.

Blackburn, H. M. (2002). Three-dimensional instability and state selection in an oscillatory axisymmetric swirling flow, *Phys. Fluids* **14**(11): 3983–3996.

Blackburn, H. M. (2003). Computational bluff body fluid dynamics and aeroelasticity, *in* N. G. Barton and J. Periaux (eds), *Coupling of Fluids, Structures and Waves Problems in Aeronautics*, Notes in Numerical Fluid Mechanics, Springer, pp. 10–23.

Blackburn, H. M., Barkley, D. and Sherwin, S. J. (2008). Convective instability and transient growth in flow over a backward-facing step, *J. Fluid Mech.* **603**: 271–304.

Blackburn, H. M., Govardhan, R. N. and Williamson, C. H. K. (2000). A complementary numerical and physical investigation of vortex-induced vibration, *J. Fluids & Struct.* **15**(3/4): 481–488.

Blackburn, H. M. and Henderson, R. D. (1996). Lock-in behaviour in simulated vortex-induced vibration, *Exptl Thermal & Fluid Sci.* **12**(2): 184–189.

Blackburn, H. M. and Henderson, R. D. (1999). A study of two-dimensional flow past an oscillating cylinder, *J. Fluid Mech.* **385**: 255–286.

Blackburn, H. M., Lee, D., Albrecht, T. and Singh, J. (2019). Semtex: a spectral element–Fourier solver for the incompressible Navier–Stokes solver in cylindrical or Cartesian coordinates, *Comput. Phys. Comm.* **245**: 106804–1–13. 50th Anniversary issue.

Blackburn, H. M. and Lopez, J. M. (2003a). On three-dimensional quasi-periodic Floquet instabilities of two-dimensional bluff body wakes, *Phys. Fluids* **15**(8): L57–60.

Blackburn, H. M. and Lopez, J. M. (2003b). The onset of three-dimensional standing and modulated travelling waves in a periodically driven cavity flow, *J. Fluid Mech.* **497**: 289–317.

Blackburn, H. M., Lopez, J. M., Singh, J. and Smits, A. J. (2021). On the Boussinesq approximation in arbitrarily accelerating frames of reference, *J. Fluid Mech.* **924**: R1–1–11.

Blackburn, H. M., Marques, F. and Lopez, J. M. (2005). Symmetry breaking of two-dimensional time-periodic wakes, *J. Fluid Mech.* **522**: 395–411.

Blackburn, H. M. and Schmidt, S. (2003). Spectral element filtering techniques for large eddy simulation with dynamic estimation, *J. Comput. Phys.* **186**(2): 610–629.

Blackburn, H. M. and Sherwin, S. J. (2004). Formulation of a Galerkin spectral element–Fourier method for three-dimensional incompressible flows in cylindrical geometries, *J. Comput. Phys.* **197**(2): 759–778.

Blackburn, H. M. and Sherwin, S. J. (2007). Instability modes and transition of pulsatile stenotic flow: Pulse-period dependence, *J. Fluid Mech.* **573**: 57–88.

Boyd, J. P. (2001). *Chebyshev and Fourier Spectral Methods*, 2nd edn, Dover, New York.

Canuto, C., Hussaini, M. Y., Quarteroni, A. and Zang, T. A. (1988). *Spectral Methods in Fluid Dynamics*, Springer, Berlin.

Canuto, C., Hussaini, M. Y., Quarteroni, A. and Zang, T. A. (2006). *Spectral Methods: Fundamentals in Single Domains*, Springer.

Canuto, C., Hussaini, M. Y., Quarteroni, A. and Zang, T. A. (2007). *Spectral Methods: Evolution to Complex Geometries and Applications in Fluid Dynamics*, Springer.

Chin, C., Ng, H. C. N., Blackburn, H. M., Monty, J. and Ooi, A. S. H. (2015). Turbulent pipe flow at $Re_\tau = 1000$: a comparison of wall-resolved large-eddy simulation, direct numerical simulation and hot-wire experiment, *Computers and Fluids* **122**: 26–33.

de Vahl Davis, G. (1983). Natural convection of air in a square cavity: A bench mark numerical solution, *Intnl J. Num. Meth. Fluids* **3**(3): 249–264.

Deville, M. O., Fischer, P. F. and Mund, E. H. (2002). *High-Order Methods for Incompressible Fluid Flow*, Cambridge University Press.

Dong, S. (2015). A convective-like energy-stable open boundary condition for simulations of incompressible flows, *J. Comput. Phys.* **302**: 300–328.

Elston, J. R., Blackburn, H. M. and Sheridan, J. (2006). The primary and secondary instabilities of flow generated by an oscillating circular cylinder, *J. Fluid Mech.* **550**: 359–389.

Funaro, D. (1997). *Spectral Elements for Transport-Dominated Equations*, Vol. 1 of *Lecture Notes in Computational Science and Engineering*, Springer, Berlin.

George, A. and Liu, J. W.-H. (1981). *Computer Solution of Large Sparse Positive Definite Systems*, Prentice–Hall.

Gottlieb, D. and Orszag, S. A. (1977). *Numerical Analysis of Spectral Methods: Theory and Applications*, SIAM.

Guermond, J. L., Minev, P. and Shen, J. (2006). An overview of projection methods for incompressible flows, *Comp. Meth. Appl. Mech. & Engng* **195**: 6011–6045.

Guermond, J. L. and Shen, J. (2003). Velocity-correction projection methods for incompressible flows, *SIAM J. Numer. Anal.* **41**(1): 112–134.

Henderson, R. D. (1999). Adaptive spectral element methods for turbulence and transition, *in* T. J. Barth and H. Deconinck (eds), *High-Order Methods for Computational Physics*, Springer, chapter 3, pp. 225–324.

Henderson, R. D. and Karniadakis, G. E. (1995). Unstructured spectral element methods for simulation of turbulent flows, *J. Comput. Phys.* **122**: 191–217.

Hughes, T. J. R. (1987). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice-Hall. (Dover edition, 2013).

Karniadakis, G. E. (1989). Spectral element simulations of laminar and turbulent flows in complex geometries, *Appl. Num. Math.* **6**: 85–105.

Karniadakis, G. E. (1990). Spectral element–Fourier methods for incompressible turbulent flows, *Comp. Meth. Appl. Mech. & Engng* **80**: 367–380.

Karniadakis, G. E. and Henderson, R. D. (1998). Spectral element methods for incompressible flows, *in* R. W. Johnson (ed.), *Handbook of Fluid Dynamics*, CRC Press, Boca Raton, chapter 29, pp. 29–1–29–41.

Karniadakis, G. E., Israeli, M. and Orszag, S. A. (1991). High-order splitting methods for the incompressible Navier–Stokes equations, *J. Comput. Phys.* **97**(2): 414–443.

Karniadakis, G. E. and Sherwin, S. J. (2005). *Spectral/hp Element Methods for Computational Fluid Dynamics*, 2nd edn, Oxford University Press.

Kernighan, B. W. and Pike, R. (1984). *The UNIX Programming Environment*, Prentice-Hall, New Jersey.

Kerr, R. M. (1985). Higher-order derivative correlations and the alignment of small-scale structures in isotropic numerical turbulence, *J. Fluid Mech.* **153**: 31–58.

Kim, J., Moin, P. and Moser, R. (1987). Turbulence statistics in fully developed channel flow at low Reynolds number, *J. Fluid Mech.* **177**: 133–166.

Kirby, R. M. and Sherwin, S. J. (2006). Stabilisation of spectral/hp element method through spectral vanishing viscosity: application to fluid mechanics modelling, *Comp. Meth. Appl. Mech. & Engng* **195**: 3128–3144.

Koal, K., Stiller, J. and Blackburn, H. M. (2012). Adapting the spectral vanishing viscosity method for large-eddy simulations in cylindrical configurations, *J. Comput. Phys.* **231**: 3389–3405.

Korczak, K. Z. and Patera, A. T. (1986). An isoparametric spectral element method for solution of the Navier–Stokes equations in complex geometry, *J. Comput. Phys.* **62**: 361–382.

Maday, Y., Kaber, S. M. O. and Tadmor, E. (1993). Legendre pseudospectral viscosity method for nonlinear conservation laws, *SIAM J. Num. Anal.* **30**: 321–342.

Maday, Y. and Patera, A. T. (1989). *Spectral Element Methods for the Incompressible Navier–Stokes Equations*, State-of-the-Art Surveys on Computational Mechanics, ASME, chapter 3, pp. 71–143.

Pasquetti, R. (2006). Spectral vanishing viscosity methods for large-eddy simulation of turbulent flows, *J. Sci. Comp.* **27**(1–3): 365–375.

Patera, A. T. (1984). A spectral element method for fluid dynamics: Laminar flow in a channel expansion, *J. Comput. Phys.* **54**: 468–488.

Piomelli, U. (1997). Large-eddy simulations: Where we stand, *in* C. Liu and Z. Liu (eds), *Advances in DNS/LES*, AFOSR, Louisiana, pp. 93–104.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (1992). *Numerical Recipes in Fortran: The Art of Scientific Computing*, 2nd edn, Cambridge University Press.

Reynolds, W. C. and Hussain, A. K. M. F. (1972). The mechanics of an organized wave in turbulent shear flow. Part 3. Theoretical models and comparisons with experiments, *J. Fluid Mech.* **41**(2): 263–288.

Rudman, M. and Blackburn, H. M. (2006). Direct numerical simulation of turbulent non-Newtonian flow using a spectral element method, *Appl. Math. Mod.* **30**(11): 1229–1248.

Sherwin, S. J. and Blackburn, H. M. (2005). Three-dimensional instabilities and transition of steady and pulsatile flows in an axisymmetric stenotic tube, *J. Fluid Mech.* **533**: 297–327.

Tadmor, E. (1989). Convergence of spectral methods for nonlinear conservation laws, *SIAM J. Num. Anal.* **26**(1): 30–44.

Temperton, C. (1992). A generalized prime factor FFT algorithm for any $n = 2^p 3^q 5^r$, *SIAM J. Sci. Stat. Comput.* **13**(3): 676–686.

Wilhelm, D. and Kleiser, L. (2001). Stability analysis for different formulations of the nonlinear term in P_N–P_{N-2} spectral element discretizations of the Navier–Stokes equations, *J. Comput. Phys.* **174**: 306–326.

Xu, C. and Pasquetti, R. (2004). Stabilized spectral element computations of high Reynolds number incompressible flows, *J. Comput. Phys.* **196**: 680–704.

Zang, T. A. (1991). On the rotation and skew-symmetric forms for incompressible flow simulations, *Appl. Num. Math.* **7**: 27–40.